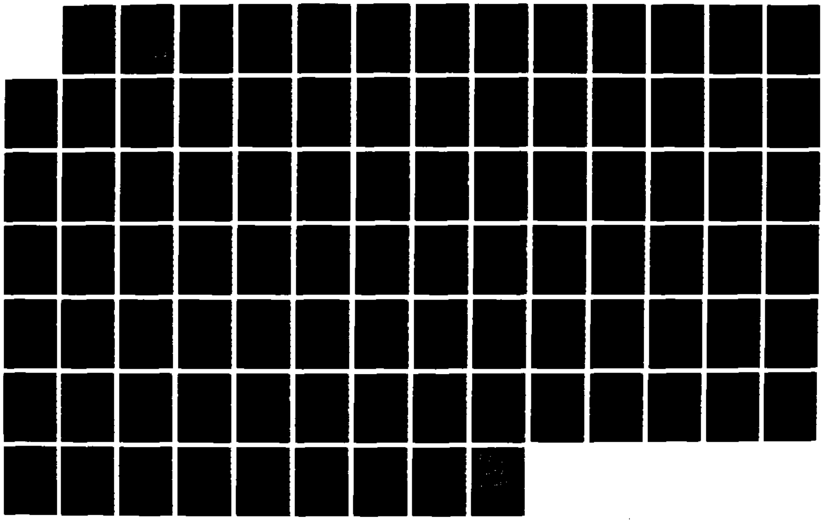
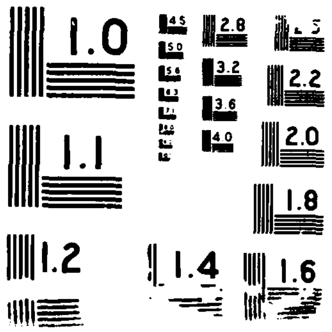
AD-A194 747 SEMANTIC TRANSFORMATIONS FOR NATURAL LANGUAGE PRODUCTION(U) UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION SCIENCES INST R WHITNEY MAR 88 1/1

UNCLASSIFIED ISI/RR-88-192 MDA903-87-C-0641 F/G 5/7 NL




1.0
2.8
2.5
4.5
5.0
5.6
6.3
7.1
8.0
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.2
1.4
1.6

ISI Research Report
ISI/RR-88-192
March 1988

(4)

University of Southern California

Richard Whitney

Semantic Transformations for Natural Language Production

INFORMATION SCIENCES INSTITUTE
213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT This document is approved for public release, distribution is unlimited.
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) ISI/RR-88-192	5 MONITORING ORGANIZATION REPORT NUMBER(S) ---------------
6a NAME OF PERFORMING ORGANIZATION USC/Information Sciences Institute 6b OFFICE SYMBOL *(If applicable)*	7a NAME OF MONITORING ORGANIZATION ---------------
6c ADDRESS *(City, State, and ZIP Code)* 4676 Admiralty Way Marina del Rey, CA 90292	7b ADDRESS *(City, State, and ZIP Code)* ---------------
8a NAME OF FUNDING / SPONSORING ORGANIZATION DARPA AFOSR 8b OFFICE SYMBOL *(If applicable)*	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA903 87 C 0641 F49620-87-C-0005
8c ADDRESS *(City, State, and ZIP Code)* [continued on next page]	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO --------------- PROJECT NO --------------- TASK NO. --------------- WORK UNIT ACCESSION NO. ---------------

11 TITLE *(Include Security Classification)*
Semantic Transformations for Natural Language Production [Unclassified]

12 PERSONAL AUTHOR(S) Whitney, Richard

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT *(Year, Month, Day)*	15 PAGE COUNT
Research Report	FROM ______ TO ______	1988, March	89

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	first-order predicate calculus; higher-order logic; meaning representation; natural language generation
09	02		

19 ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This thesis presents a technique for connecting the representations produced by a parsing and understanding component to an English language generation component. ~~We describe~~ transformations are applied to higher-order predicate calculus expressions to produce equivalent first-order forms suitable for driving the text generator. These transformations are expressed as axiom schemata. Their adequacy is demonstrated by a paraphrase task. ~~In~~ this task ~~we~~ transforms higher-order expressions representing English queries into first-order expressions, and then ~~use~~ these in our generation system to produce accurate English versions of the original queries.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] [✓ SAME AS RPT.] ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Sheila Coyazo Victor Brown	22b TELEPHONE *(Include Area Code)* 213-822-1511 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

8c. ADDRESS continued

Defense Advanced Research Projects Agency (DARPA)
1400 Wilson Boulevard
Arlington, VA 22209

Air Force Office of Scientific Research (AFOSR)
Bolling Air Force Base, Building 410
Washington, DC 20332

ISI Research Report
ISI/RR-88-192
March 1988

University of Southern California

Richard Whitney

Semantic Transformations for Natural Language Production

INFORMATION SCIENCES INSTITUTE

213/822-1511

4676 Admiralty Way/Marina del Rey/California 90292-6695

This research was sponsored in part by the Defense Advanced Research Projects Agency under contract no MDA903-87-C-0641, and in part by the Air Force Office of Scientific Research under contract no. F49620-87-C-0005 Any opinions,findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the sponsors

ACKNOWLEDGEMENTS

The author is especially grateful to William Mann and Norman Sondheimer for creating the fertile environment in which this work could occur. Robert Albano, Eduard Hovy, and Robert Kasper all provided helpful comments on earlier versions of this report. Valuable discussion of many of these ideas took place with members of the staff of Bolt, Beranek, and Newman, Inc., including Damaris Ayuso, Erhard Hinrichs, Ralph Weischedel, and Remko Scha.

This research is supported by the Defense Advanced Research Projects Agency under Contract No. MDA903 87 C 0641 and by the Air Force Office of Scientific Research under F49620-87-C-0005

Table of Contents

List of Figures

1. Introduction

The text-generation task is often seen as a peripheral concern in natural language systems--the designs of most systems, and the bulk of their resources, are committed to comprehending and representing rather than to reporting. This paper shows how the generation process can be a central part of a natural language system by sharing the top-level meaning representation language. We have developed a system that derives its expressive demands from the ongoing flow of information in a natural language question-answering system, where the dominant process is that of accepting English questions and reducing them to the level of relational database queries.

Why should this sharing of resources be of interest? Just as the kinds of syntactic and lexical information that are useful for parsing differ from those useful for generation (witness the nonexistence of a practical, large-scale, bidirectional grammar), so the kind of representation produced by parsers will differ from the kind required for successful generation. This became especially clear in our work, in which we joined a text generator to an independently developed parser. While the parser produces higher-order forms and multiple readings of individual sentences, our generator uses first-order predicate calculus forms and employs semantic distinctions that make linguistic sense (according to the categories of a systemic grammar). In translating the representations produced by the parser into the representations required by the generator, we arrived at useful insights about the nature of the two processes and about the suitability of representation languages to express the information requisite for the generation task.

The vehicle for this work is the Penman system, one of whose goals is to imbed the Nigel text generation grammar [Mann & Matthiessen 83], in the

Janus[1] natural language input/output processor to produce a comprehensive natural language text delivery system. At this time, the Nigel generator is able to paraphrase a database query, given in English, from the representation produced by the parsing component. That is, we take the logical form representing the input sentence as a specification of the material to be expressed as output in English. Since the parser may produce multiple forms as candidate readings for the input sentence, paraphrase provides the utility of showing to the user the ambiguous character of his query by expressing the alternate readings as distinct sentences in English. We have also been able to demonstrate, in a limited way, the ability in Janus to blend the database output with the representation for the English query in order to produce an English-language answer. Although we are still working to develop these applications of the generation process within the question-answering environment, our progress gives us some reason to expect that the Penman component will be able to generate competent English from arbitrary demands expressed in the representational formalism of the understanding component.

A first-order predicate calculus (FOPC) meaning representation language, and its realization in the NIKL/KL-TWO knowledge representation and reasoning system [Kaczmarek et al. 86, Vilain 85], provide a sound technique for interpreting the queries produced by the Nigel sentence generator concerning the output it is forming [Sondheimer & Nebel 86]. The success of that logical form/knowledge base design also demonstrates the adequacy of the grammar to accept and realize the information conveyed by expressions of

[1]The Janus natural language interface is a joint effort between Bolt, Beranek, and Newman, Inc. and USC/ISI within the natural language technology component of the Strategic Computing Initiative [Walker et al. 85]. The Janus system is implemented in Common Lisp and runs on Symbolics and Texas Instruments workstations.

FOPC. However, the design employs certain key concepts that are oriented to the grammar's view of the world. These concepts include a fourfold distinction between events, relations, verbal actions, and perceptions--a distinction that reflects the grammar's alternatives as it lays out the basic structure of a sentence. The logical form/knowledge base design also requires that certain instances of the concepts in that fourfold distinction be reified. We will deal more concretely with the role these features play in the generation component as we show how simple relational predications are mapped into logical forms suitable for responding to the inquiries of the grammar. Transformations at the level of concepts, and instances of them, are the subject of Section 2 of this paper.

We will also show how we transform a higher-order language into our FOPC notation. This higher-order language is a predicate calculus extended by operators for intension, tense, modality, lambda abstraction, and sets. It is produced by the understanding component of the Janus system and serves as a meaning representation from which further processing proceeds in two directions: first, by one series of mappings, to the construction of a query to the relational database; and second, by another series of mappings, through the grammar, to English output. In that second series of mappings, a major step is the move from the higher-order forms to the FOPC meaning representation for interpreting the grammar's inquiries. This process comprises the main technical subject matter of this report. Thus, in addition to showing how language generation can be a useful component of a natural language processing system, our work also confirms the choice of the higher-order meaning representation language as the epistemological currency of the Janus system.

2. A Simple Example: "Frederick is C2"

In this section we develop transformations, central to the generation process, that take place at the level of FOPC[2]. Our motive is to create a direct mapping between terms of the logical form and the "hubs" of the grammar. (A "hub" is a symbol shared by the grammar and its environment, that denotes an element in the text.) Each hub is an instance of a concept represented in the NIKL model, and the collection of sentence elements are related by NIKL roles. Our transformations at this level are controlled by a collection of axiom schemata, whose instantiations depend on where the concepts and roles involved fall in the NIKL model. These schemata make explicit the assumptions involved in our linguistic view of the conceptual model of the knowledge base; they also provide a focal point for a precise characterization of the truth conditions for formulas in our meaning representation language.

2.1. Relations as Entities

We begin with an example of what we take to be a simple relational predication: (UNITS.OVERALL.READINESS FREDERICK C2).[3] The constants FREDERICK and C2 denote the individuals participating in the UNITS.OVERALL.READINESS relation. As atomic individual terms, these symbols identify to Nigel elements of the intended text. On the other hand, the UNITS.OVERALL.READINESS relation itself is not represented as an in-

[2]In actuality, these transformations range over expressions that are subsumed by the higher-order language; they could, however, be re-formulated in a sub-FOPC quantifier-free language consisting simply of relations and individual terms.

[3]The domain-specific part of our current work is based on a database of naval ships and facts concerning their location and condition. This example relation might be understood on the model of the relation between objects and colors: (COLOR BETELGUESE RED) represents the same fact as that expressed by the sentence "Betelguese is red." Here, UNITS.OVERALL.READINESS represents a relation between the ship Frederick and its readiness condition, and C2 is a term encoding a level of operability.

dividual term, but it does correspond to one of the hubs in the grammar's realization of the corresponding text. We need a way to refer to this particular instance of the UNITS.OVERALL.READINESS relation, in order to answer Nigel's inquiries concerning the character of the hub that relates the two elements FREDERICK and C2. Typical characteristics include the following: (a) It is a relation, and not an event, verbal act, or perception. (b) It is a relation of a particular kind, that of property ascription. (c) The things it relates are denoted by the symbols FREDERICK and C2, and in that order. In order to qualify this specific instance of the UNITS.OVERALL.READINESS relation, we need to reifiy it--in other words, to create an entity that is an instance of the UNITS.OVERALL.READINESS relation. We then identify the participants in the original relation as involved in canonical relations ("domain" and "range") with the newly created entity. We use the axiom schema shown in Figure 2-1.[4]

```
(ALL X Y (UNITS.OVERALL.READINESS X Y) =>
         (EXIST U (UNITS.OVERALL.READINESS U)
          (AND (DOMAIN U X)
               (RANGE U Y))))
```

Figure 2-1: Axiom for UNITS.OVERALL.READINESS relation

Applying this to the original predication yields the expression in Figure 2-2. Our transformed representation now provides a particular individual term, U,

[4]Our logical language is marked by breaking quantified expressions into range and predication components with the following equivalences:

```
(EXIST X (P X) (Q X)) <=> (EXIST X (AND (P X)(Q X))).
(ALL X (P X) (Q X)) <=> (ALL X (IMPLIES (P X)(Q X))).
```

To represent the transformation, we show the input expression in the axiom as the left-hand side of the symbol '[illegible]' and the result as its right-hand side. Although the notation suggests a relation of implication, we actually take the two sides to be equivalent. As a matter of practice, we invoke the equivalence only in the direction indicated; as a matter of principle, we understand the converse to hold.

```
(EXIST U (UNITS.OVERALL.READINESS U)
 (AND (DOMAIN U FREDERICK)
      (RANGE U C2))).
```

Figure 2-2: An instance of the UNITS.OVERALL.READINESS relation

to be an instance of the UNITS.OVERALL.READINESS relation. As such, U occupies a particular position in the hierarchy of relations known to NIKL. In generating, the Nigel grammar will ask a series of questions about U, such as whether it represents an event, a perception, or a verbal action. Its place in the NIKL hierarchy will furnish the answer, distinguishing this particular instance as relational (and therefore distinct from events, verbal actions, and perceptions) and, more particularly, as a kind of property ascription, i.e. a relation between a thing and some property or quality of it. The reified relation also provides a node in the NIKL concept hierarchy with which we associate appropriate lexical information. Furthermore, placing the terms involved in the relation (FREDERICK and C2) into distinguished roles, and identifying them to Nigel as hubs, causes them each to become the object of a further series of inquiries in order for Nigel to realize their expression. Thus, the introduction of new terms and concepts allows us to use the hierarchical reasoning component of KL-TWO to answer straightforwardly Nigel's classificatory inquiries.

In practice, we generalize the schema shown in Figure 2-1 to apply to any relation known to the NIKL hierarchy (Figure 2-3)[5].

[5]Here, NKSUPERC? is a predicate that is true if the relational predicate name, P, is known to be a member of the hierarchy of two place relations.

```
(ALL P X Y (AND (NKSUPERC? P 2-PLACE-RELATION)
                (P X Y)) =>
                (EXIST Z (P Z)
                 (AND (DOMAIN Z X)
                      (RANGE Z Y))))
```

Figure 2-3: The general schema for two-place relations

2.2. Occurrences

As it is, the reified relation is incomplete; it lacks a time. Therefore, we augment our transformation of the initial form in one more way: through the indication of an occurrence and the attachment of a time role to it. Our notion of an occurrence is akin to the eventualities of Hobbs [Hobbs 85]. From this perspective, marking a relation as occurring moves it from the realm of *possible* entities (which is the domain over which our quantifiers range) to the realm of *actual* entities. But there is a simultaneous linguistic motive for the use of this additional concept, and that is to mark the difference between events or states of affairs that are to be expressed as occurring during a determinate time (whether known or not) and those that are expressed with no commitment to any such occurrence. The indication of an occurrence is a way of representing that the process holds, as opposed to processes that need not--or indeed cannot--hold, as the process that John wants in the situation expressed by the sentence "John wants to square the circle." The role of occurrence marks a difference in the readiness status of the ships in the situations expressed by the sentences "The C2 ship is in San Diego" and "The ship which was C2 is in San Diego" through its absence or its presence (respectively)[6]. Different occurrences may involve the same relation at dif-

[6]This comparison is a bit misleading, since in some cases we may choose to express an occurrence with nominal time characteristics (i.e. occurring in the present) by a simple modifier rather than by a relative clause. In general, we should think of an occurrence as something that may require a tensed verb to be fully expressed.

ferent times, and so this formulation makes it possible to express the notion of recurrent events or relations.

Our transformation gives explicit representation to these factors through the additional concept of OCCURRENCE, and through that roles that mark the connection between the occurrence and (a) the event or relation (RECORDS) and (b) the time of the occurrence (TIMEOFOCCURRENCE). The final result of our transformations on the original form is given in Figure 2-4.[7]

```
(EXIST W (OCCURRENCE W)
 (AND (TIMEOFOCCURRENCE W (ETA T (PRESENT T)))
      (EXIST U (UNITS.OVERALL.READINESS U)
       (AND (RECORDS W U)
            (DOMAIN U FREDERICK)
            (RANGE U C2)))))
```

Figure 2-4: Transformation of the readiness relation

The concepts and relations ocurring in the meaning representation of Figure 2-4 are asserted in the KL-TWO knowledge base as a verbalization graph, shown in Figure 2-5. The nodes are the terms of the formula and serve as hubs for the grammar's inquiries; a concept is associated with a node in the graph when it is predicated of the term in the logical form. The arcs are labeled with the relations between the terms. We can view the generation process as entering this graph at an identifiable point--the process occurrence--and proceeding across the arcs to the connected nodes, each of which will in turn be realized by the grammar in a particular form of expression. We expect certain properties from the verbalization graph: notably, that it be connected and, except for a limited determinate set of primitives, that every con-

[7]The ETA operator is based on that of Habel [Habel 82]. This operator signals an individual not fully identified by the predication contained in the scope of the operator. In this case, we assume the occurrence occupies some unspecified interval that overlaps the present.

```
                        (OCCURRENCE W)
                          /        \
                   RECORDS    TIMEOFOCCURRENCE
                     /              \
      (UNITS.OVERALL.READINESS U)  (PRESENT T)
                 /   \
            DOMAIN  RANGE
              /       \
        FREDERICK     C2
```

Figure 2-5: Verbalization graph representation of logical form

cept and relation be defined in the NIKL hierarchy. Both of these concerns arise from the constraint that the generation system must express everything represented in the source formula. Without connectivity, we would not expect to relate every node to the starting point; without its being known to the NIKL hierarchy, we would not know what to say to express a node. The quantifier information from the original form is retained separately, and serves to answer a small number of the grammar's inquiries concerning determination and number in referring expressions.

3. Transforming Higher-Order Logic Forms to FOPC

FOPC is satisfactory for many representational tasks. In particular, we believe it to be adequate to represent linguistic demands for the purpose of generation. The formalization of expressions of natural language in higher-order logic is a fundamental tenet of some recent and sophisticated research programs in the philosophy of language [Montague 74]. The utility of such a logic in the transformation of natural language queries to the level of database retrieval is an object of current development in the Janus system. The parsing and understanding component of Janus uses a higher-order language as its primary representational formalism, from which first-order expressions are derived. Those first-order forms are, however, tailored to their eventual reduction to a relational database query. Consequently, they embody primitive notions appropriate to database access. In our generation component, which is adequately governed by FOPC, we transform the higher-order forms to FOPC forms tailored to the generation task.

The expression in Figure 3-1 is an example of a form from the higher-order language[8, 9]. It is intended to represent the situation expressed by the sentence "What ships are harpoon capable?"

```
(QUERY
 ((INTENSION
   (PRESENT
    (INTENSION
     (PRED-TO-SET
      (LAMBDA (X) (POWER VESSEL)
       (HARP.CAPABLE.SHIP X))))))
 TIME WORLD))
```

Figure 3-1: Sample higher-order form

The atom QUERY represents the nature of the speech act of the incoming expression. It applies to the extensional entity that is the result of evaluating the outer intensional expression (marked by the atom INTENSION,) with respect to a particular time and a particular possible world. (In this form, the atoms TIME and WORLD denote the present time and the actual world.) The speech act operator itself is always carried over into our FOPC meaning representation. However, it has no effect on our formation of the verbalization graph from the FOPC formula because we treat the speech act as an aspect of the discourse situation and distinct from the predicate calculus formula itself. Immediately within the outer INTENSION operator is the tense

[8]The precise syntax and semantics of this language are presented in an internal BBN technical memorandum by Erhard Hinrichs; a condensed version appears as Appendix A.

[9]The examples presented here are the actual output from the parser of the preferred reading for the English input.

(or time) operator, PRESENT. It applies to the inner intensional expression, which consists of a set-forming operator, PRED-TO-SET. This operator in turn takes a generalized predicate given by a form for lambda abstraction that is marked by the following: (a) the atom LAMBDA; (b) the variable of abstraction, X; (c) an expression, (POWER VESSEL), that restricts the range of the variable X to the power set of the set of vessels; and, (d) a formula open with regard to the variable X.

Within the context of a database query system, we should note that such higher-order forms will ultimately reduce to database queries at a simple relational level. The functions for which a higher-order language is especially useful, such as quantifying in intensional contexts, are of limited utility in representing queries directed at an extensional database. Thus, while the utility of a higher-order language in the parsing process is evident, we believe that, given our current application, this higher-order representation will not present linguistically relevant information that we would not be able to transform successfully into our FOPC meaning representation for the generation task. In the following section, we will discuss other contexts suitable for future research. The remainder of this section deals with specific examples of how we achieve the conversion from higher-order language to FOPC.

3.1. Tense

The source language communicates essential time information by way of tense operators. In our interpretation, we identify the occurrence of a process (i.e. an event or a relation[10]) for each tense operator. The tense operator also specifies the time of occurrence of the process, which we express

[10]Our domain model does not currently include any concepts subsumed by perception or verbal action.

as a relation between the occurrence and a time period.

As the identification process proceeds, we search for that process most immediately within the scope of the tense operator. If the process is an event, the actor, actee, and beneficiary participants are already identified explicitly in the form. If the process is a relation, we apply the transformation discussed in the preceding section. If neither an event nor a relation is found, we assume the process to be one of class ascription, the transformation of which is discussed in Section 3.3. The resulting form is a transformation of the original with a process occurrence in the RECORDS relation to the identitifed process and a corresponding time of occurrence in place of the tense operator. Basically, we create the process occurrence envelope like the one that was added to the reified relation of Figure 2-4, with the time of occurrence coordinated with the tense operator.

Figures 3-2 and 3-3 show the input and output expressions corresponding to the English sentence "The ship which was C3 is C2."

```
(ASSERT
 ((INTENSION
   (PRESENT
    (INTENSION
     (UNITS.OVERALL.READINESS
      (IOTA X VESSEL
       (PAST
        (INTENSION
         (UNITS.OVERALL.READINESS X C3))))
     C2))))
 TIME WORLD))
```

Figure 3-2: Higher-order form for "The ship which was C3 is C2."

The primary modifications to the incoming form are: (a) reification of the readiness relation in its two instances; (b) insertion of process occurrences recording those relations; and (c) the time of occurrence reflecting the tense

```
(ASSERT
 (EXIST U (OCCURRENCE U)
  (AND (TIMEOFOCCURRENCE U (ETA T (PRESENT T)))
       (EXIST V (UNITS.OVERALL.READINESS V)
        (AND (RECORDS U V)
             (DOMAIN V
              (IOTA X
               (AND (VESSEL X)
                (EXIST W (OCCURRENCE W)
                 (AND (TIMEOFOCCURRENCE W (ETA T (PAST T)))
                      (EXIST Y (UNITS.OVERALL.READINESS Y)
                       (AND (RECORDS W Y)
                            (DOMAIN Y X)
                            (RANGE Y C3))))))))
             (RANGE V C2))))))
```

Figure 3-3: First-order equivalent for "The ship which was C3 is C2."

operator of the incoming meaning representation. In this example, the original form had intension operators paired with tense operators, as required by the semantics of that language. We dismiss the intension, since its effect for us is captured in relating the time information to the UNITS.OVERALL.READINESS relation through the process occurrence. The intension is fully recoverable in back-translation. Also, the outermost INTENSION operator, coupled with the TIME and WORLD indices, is a form required as the parameter to the speech act operator. We dismiss these as well, since they too are recoverable in back-translation.

We have also applied a minor syntactic transformation for the term governed by the IOTA operator. The syntax for the incoming formula allows for a predicate restricting the variable bound by the IOTA; the syntax for our language does not. The rule we apply in such cases is expressed in Figure 3-4[11].

[11]Schemata of this form represent a kind of hybrid formulation since they contain components from distinct languages. Also, the predicate letter R should be understood to represent an arbitrary form open with respect to the variable bound by the operator.

```
(ALL X   (P (IOTA X Q (R X))) =>
         (P (IOTA X (AND (Q X)(R X)))))
```

Figure 3-4: Schema for transforming IOTA

3.2. Abstraction

Lambda abstraction is employed as a generalized predicate in the higher-order language we deal with. The syntax and semantics of that language require the operand of the set constructor and the range-restricting operand of the universal and existential quantifiers to be functions from individuals to truth values rather than arbitrary formulas, which are truth-valued entities. The true higher-order power of abstraction over predicates is not currently invoked. We perform a syntactic transformation of the operator to ensure connectivity in the resulting verbalization graph. For instance, the higher-order form in Figure 3-5 represents the English sentence "Are any C2 ships harpoon capable?"

```
(QUERY
 ((INTENSION
   (PRESENT
    (INTENSION
     (EXIST X
      (LAMBDA (Y) VESSEL
       (UNITS.OVERALL.READINESS Y C2))
  (HARP.CAPABLE X)))))
 TIME WORLD))
```

Figure 3-5: Form with lambda abstraction

In this form, the LAMBDA sub-form represents a complex restriction (ships in an overall readiness relation with C2) on the variable X, which is governed by the existential quantifier EXIST.

Where the higher-order language requires that the type of the restriction on the variable governed by EXIST be a function from individuals to truth values, our language allows complex expressions (arbitrary formulas, open with regard to the variable bound by the quantifier) in the range portion of a quantified formula. Therefore, translation from such higher-order forms is basically a matter of rewriting with the appropriate variable replacement. In our transformation, the lambda abstraction is absorbed into the restriction on the variable governed by the existentially quantified sub-formula. This operation is represented by the schema in Figure 3-6.

```
(ALL P Q R (EXIST X (LAMBDA (Y) P (Q Y))(R X)) =>
           (EXIST X (AND (P X)(Q X))(R X)))
```

Figure 3-6: Schema for abstraction conversion

This schemea is applied to the existentially quantified sub-form of the higher-order form of Figure 3-5; the resulting form is shown in Figure 3-7.

```
(EXIST X (AND (VESSEL X)
              (EXIST Y (UNITS.OVERALL.READINESS Y)
               (AND (DOMAIN Y X)
                    (RANGE Y C2))))
 (HARP.CAPABLE X))
```

Figure 3-7: Transformed lambda abstraction

Additional operations will imbed this sub-form into the appropriate process occurrence. As another instance, we saw the sub-formula of Figure 3-8

```
(PRED-TO-SET
 (LAMBDA (X) (POWER VESSEL)
  (HARP.CAPAPBLE.SHIP X)))
```

Figure 3-8: Higher-order representation for the set of harpoon capable ships

representing the set of harpoon capable ships, in the representation for the sentence "What ships are harpoon capable?" Here the abstraction operation again serves as a generalized predicate and is handled in a fashion very similar to the previous one. We will see the full transformation of this form in the next section.

3.3. Sets and Class Ascription

Our first-order language provides for explicit representation of sets with the intensional and extensional set-forming operators, SET-OF and SET-WITH. The first denotes a set by applying a formula to the individuals in the universe of discourse, and collecting those that satisfy the formula: (SET-OF X (VESSEL X)). The second denotes a set by enumerating the individuals in it: (SET-WITH FREDERICK MIDWAY). Our conceptual model provides a SET concept, a relation MEMBER-OF, a relation TYPE-OF, and a relation CARDINALITY.

In some cases, expressions denoting sets are present in the higher-order form given to the generation process. In those cases, we perform appropriate syntactic revision to produce an equivalent form in the notation of our own representation language. However, some transformations we apply to an incoming form cause us to introduce expressions denoting sets.

Both cases of set construction are exemplified in the meaning representation for the sentence "What ships are harpoon capable?" (Figure 3-1). The PRED-TO-SET operator will appear in our first-order form as the operator SET-OF; the range restriction of the LAMBDA expression, (POWER VESSEL), will be reduced to the operative predicate VESSEL, and conjoined to the expression representing the predicate abstracted by LAMBDA. In the course of transforming this sentence, the tense operator PRESENT, which lies outside the PRED-TO-SET operator, must be accounted for with the occurrence of a process. The incoming representation has no process--neither an event, like DEPLOYMENT, nor a two-place relation, like UNITS.OVERALL.READINESS. In such a case, where it is necessary to identify a process recorded by a required process occurrence, we create a reified class ascription relation. The higher-order forms for such sentences, like ordinary predicate calculus expressions, capture the relation between the individual and the class of that it is a member through simple predicate attribution (application). As in the case of two-place relations, our approach is to identify the attribution as a relation of a specifiable kind. One of the entities involved in the relation is an individual term; the other is a term denoting the set of things which share the attributed property. The relation between them is then one of class ascription. Restricting its employment to occurrences of processes prevents the possible unlimited regress--due to the recurrence of the predicate attribution--implicit in the axiom schema of our transformation for one-place predications (see Figure 3-9). This schema is instantiated in this case by the transformation displayed in Figure 3-10.

```
(ALL P X (P X) =>
        (EXIST Y (CLASSASCRIPTION Y)
                 (AND (DOMAIN Y X)
                      (RANGE Y (SET-OF Z (P Z))))))
```

Figure 3-9: Reification of one-place predications

```
(ALL X (HARP.CAPABLE.SHIP X) =>
      (EXIST Y (CLASSASCRIPTION Y)
               (AND (DOMAIN Y X)
                    (RANGE Y
                     (SET-OF Z (HARP.CAPABLE.SHIP Z))))))
```

Figure 3-10: Transformation of predicate to class ascription

Note that we identify a new individual, in the form of a set intensionally defined by the given predicate, to serve as the range of the class ascription relation.

The reification of the class ascription predication, triggered by the tense operator in the incoming meaning representation, precedes the transformation of the interior PRED-TO-SET operator. In combination, these two operations yield the first-order representation of the original higher-order form shown in Figure 3-11.

```
(QUERY
 (SET-OF X
  (AND (VESSEL X)
       (EXIST Y (OCCURRENCE Y)
        (AND (TIMEOFOCCURRENCE Y (ETA T (PRESENT T)))
             (EXIST Z (CLASSASCRIPTION Z)
              (AND (RECORDS Y Z)
                   (DOMAIN Z X)
                   (RANGE Z
                   (SET-OF W (HARP.CAPABLE.SHIP W))))))))))
```

Figure 3-11: Transformation of "Which ships are harpoon capable?"

This transformation of the PRED-TO-SET operator makes use of the schema shown in Figure 3-12.

```
(ALL P Q R X (R (PRED-TO-SET (LAMBDA (X) (POWER P) (Q X)))
          => (R (SET-OF X (AND (P X) (Q X))))))
```

Figure 3-12: Schema for transforming set operators

Our translated form indicates the occurrence of a "wh-" question by translating the set constructor PRED-TO-SET as SET-OF, and installing this term operator as the top-level form within the speech act context. The class predication of the lambda abstraction is identified as the underlying process and, as such, it is treated as a reified class ascription. What was for the LAMBDA operator a predicate restricting its range (namely VESSEL) we conjoin with the sub-formula expressing the process occurrence, which is free at the DOMAIN role with regard to the variable bound by the outermost set constructor. An exactly analogous procedure is applied to the class predication in the higher-order form for the sentence "Are any C2 ships harpoon capable?"

We also consume and produce set operators in two other contexts: (a) where the intention is to represent the singleton set corresponding to definite description, and (b) where the IOTA operator is intended to identify a particular set. Both situations are encountered in the higher-order form for the sentence "Where are the C4 ships?" (Figure 3-13).

```
(QUERY
 ((INTENSION
   (PRESENT
    (INTENSION
     (PRED-TO-SET
      (LAMBDA (X) LOCATION.IN.SPACE
       (LOCATION.OF
        (IOTA Y (POWER VESSEL)
         (UNITS.OVERALL.READINESS Y C4))
        X))))))
  TIME WORLD))
```

Figure 3-13: Higher-order form representing "Where are the C4 ships?"

Since the restriction on the variable of the LAMBDA expression is LOCATION.IN.SPACE, and not the power set of the predicate, the resulting set is a singleton which we regard as an individual governed by a definite description. In the IOTA expression, the range of the variable is the power set of VESSEL, and so requires a set operator in our notation. Our first-order equivalent is shown in Figure 3-14. The two schemata employed in this transformation are shown in Figure 3-15; the range component P, of the transformation of the PRED-TO-SET operator is understood not to be a power set operator.

3.4. Operators of Intension

The use of operators of intension in the higher-order meaning representation language is derived largely from the work of Professor Richard Montague [Montague 74]. He addressed several thorny philosophical problems in the program of obtaining formal representations of English expressions consistent with preserving the validity of arguments involving those expressions

```
(QUERY
 (IOTA X
  (AND (LOCATION.IN.SPACE X)
       (EXIST O (OCCURRENCE O)
        (AND (TIMEOFOCCURRENCE O (ETA T (PRESENT T)))
             (EXIST L (LOCATION.OF L)
              (AND (RECORDS O L)
                   (DOMAIN L
                    (SET-OF V
                     (AND (VESSEL V)
                          (EXIST R
                           (UNITS.OVERALL.READINESS R)
                           (AND (DOMAIN R V)
                                (RANGE R C4)))))
                   (RANGE L X))))))))
```

Figure 3-14: First-order form representing "Where are the C4 ships?"

```
(ALL P Q R X (R (PRED-TO-SET (LAMBDA (X) P (Q X)))) =>
             (R (IOTA X (AND (P X) (Q X))))

(ALL P Q R X (R (IOTA X (POWER P) (Q X))) =>
             (R (SET-OF X (AND (P X) (Q X)))))
```

Figure 3-15: Additional schemata for set transformations

(the paradox of the name relation being one example of this sort of problem). A full explanation of Montague's intensional logic is clearly beyond the scope of this report, but it is important to point out the role of these operators in the context of natural language processing in a question answering system.

Intensions are functions from indices to extensions. Let us explain this succinct definition by elaborating the terms "indices" and "extensions". The indices relevant to understanding the meaning of a sentence (and consequently evaluating its truth or falsity) might include its author, its audience, the time of its deliverance, the place of its deliverance, the set of objects that can be pointed at (i.e. potential targets of words like 'this' and 'that'), the segment of discourse of which this sentence is a part, and a possible world.[12] Extensions are truth values, in the case of sentences; collections of objects, in the case of common nouns; or objects, in the case of names.[13] For example, in a particular sentence, an intension carries us from a collection of relevant indices to a truth value. For a particular name, an intension carries us to a specific object (if such exists) to which the name refers. In the higher order language we are here considering, the indices adopted consist of the time the sentence is expressed and the relevant possible world. Typically, these assume their default values: the present time and the actual world.

We see the intension operator in two different kinds of places in the higher-order meaning representations. First, it appears in a context that calls for evaluation of the intensional form--specifically, as the outermost operator of an expression paired with time and place indices immediately within the scope of a speech act operator. Speech act operators take extensions as their sole argument; and the intension evaluated at a time and possible world results in an extension, which is typically, a truth value or a set of entities from the domain of discourse. If the speech act operator indicates a question, then the value is returned as the answer. If the speech act operator indicates an assertion, then the value is recorded in appropriate form in the knowledge base.

The second place we see the intension operator is in a context where

[12]I say the set of indices "might" include the items of this list because the actual choice depends upon the underlying theory. A Montague style meaning representation is founded on possible world semantics; situation semantics [Barwise & Perry 83, Cooper 87] departs from that tradition. For a full account of the role of indices in semantics of the style of Montague, see [Lewis 72].

[13]Wh-interrogatives that may be answered by producing a set of values ("Where is the cook?" "The kitchen or the pantry") we also view as having a set-valued extension.

evaluation of the form bound by the intensional operator is to be controlled by its presence, viz., as the outermost operator of the expression governed by a tense operator or a sample operator. With these operators, the intent of the intension operator is to restrain evaluation rather than to effect it. The tense operator causes a certain specifiable transformation of the entity within its scope, a transformation in which time indices representing the sense of the tense operator are attached to predicates within the form.

This latter role is especially apparent in the higher-order forms we have seen here, for it facilitates the reduction of these forms to the level of database queries. For our purposes, such control is not necessary, since we do not evaluate these forms. The words chosen to express those concepts where the intension would be made explicit in the logical form (e.g. belief) adequately express the intensional character of the context. While the presence of these operators in the higher-order language has ample theoretical justification, in practice their use has been limited to entirely predictable contexts. Our treatment of these operators has always been to delete them from the contexts of speech act operators and tense operators. However, we anticipate that in the future the time index will be mapped onto each individual predication as part of the evaluation of the tense operators. The result will be a form whose logical complexity, as measured by the character of the operators involved, will be reduced at the cost of adding another argument (a time) to each predication. In any case, sufficient information is provided to generate adequately expressive and accurate English text.

3.5. Modal Operators

The higher-order meaning representation language includes an operator for recording features of English sentences conveying modalities of truth, such as the matter of definition represented in the sentence "A destroyer is necessarily a vessel." We believe the framework provided by the notion of occurrence affords an adequate re-representation of such modal operators, and is able to express additional modal concepts, such as the deontic modes (the modes of obligation) or epistemic modes (i.e. whether the truth value of a proposition is known or unknown). Suppose that distinct modal operators, such as epistemic necessity and obligation, are represented as relations in our NIKL hierarchy and are classified, as are other concepts, in accord with the grammar's distinctions. Then we can view a higher-order form in which a modal operator applies to a predication as equivalent to a form in FOPC where the coordinate process occurrence is involved in a modal relation with a particular modal quality. So if the higher-order form represents the fact that destroyers must be ships--a matter of definition in our model--in the form displayed in Figure 3-16, then the re-representation of that fact in our meaning representation would be as shown in Figure 3-17.

```
(ASSERT
 ((INTENSION
   (PRESENT
    (INTENSION
     (NECESSARILY
      (ALL X DESTROYER
        (VESSEL X))))))
 TIME WORLD))
```

Figure 3-16: A higher-order form with a modal operator

That is, there is a relation of epistemic modality between the process occurrence and a particular modal value, which is necessity. Different forms of

```
(ASSERT
 (EXIST X (TRUTHMODALITY X)
  (AND (EXIST Y (OCCURRENCE Y)
        (AND (TIMEOFOCCURRENCE Y (ETA T (PRESENT T)))
             (EXIST Z (CLASSASCRIPTION Z)
              (AND (RECORDS Y Z)
                   (DOMAIN Z (SET-OF U (DESTROYER U)))
                   (RANGE Z (SET-OF V (VESSEL V))))))
             (DOMAIN X Y)
             (RANGE X (IOTA W (NECESSITY W))))))))
```

Figure 3-17: First-order representation of the modality.

modality, such as occurrences obligatory or forbidden, would involve a non-epistemic modal relation, though the quality may remain the same. Thus, to represent the sense of obligation in "Kennedy must be deployed," we would anticipate appropriate generation from a form like that of Figure 3-18.

```
(ASSERT
 (EXIST X (DEONTICMODALITY X)
  (AND (EXIST Y (OCCURRENCE Y)
        (AND (TIMEOFOCCURRENCE Y (ETA T (PRESENT T)))
             (EXIST Z (DEPLOYMENT Z)
              (AND (RECORDS Y Z)
                   (ACTEE Z KENNEDY))
             (DOMAIN X Y)
             (RANGE X (IOTA W (NECESSITY W))))))))
```

Figure 3-18: First-order representation of deontic modality

4. Paraphrase and Answer Generation

To demonstrate the adequacy of the transformations described in the preceding sections, we have applied our generation system to two tasks, paraphrase and answer generation.

4.1. Paraphrase

The Janus natural language question-answering system produces a meaning representation for an English query. The Penman component reduces the higher-order representation for a query to a first-order form and uses that form as input to the generator. The generator, in turn, produces an English question as a paraphrase of the original query. This process can serve two functions. First, it provides a check on the overall system. If the paraphrase differs markedly from the original in meaning, then there has been a mistake in processing: either the parser has misrepresented the input, or the reduction to first-order form has failed, or the generator has incorrectly processed its input. Second, it offers a means of communicating with the user when the English question does not reduce to a single database query. Since the parser will produce multiple representations for English input that it finds to be ambiguous, the possible readings can be returned to the user in disambiguated English--sentences generated from the distinct representations--and the user can then choose among them, or reformulate the question.

The latter function is clearly of more utility to the user, and we will give some examples of its application. Consider the question "Which destroyer that is in the battle group of a carrier that is C1 is in the Indian Ocean?" The parser obtains for this question the two distinct representations shown in Figure 4-1. The different readings reflect the possibility that the relative clause "that is C1" can apply to either the destroyer or the carrier. The mapping to first-order form produces correlate representations, one of which is shown in Figure 4-2. The English sentences resulting from applying the generator to these forms are "Which C1 destroyer that is in the battle group of a carrier is in the Indian Ocean?" and "Which destroyer that is in

```
(QUERY
 ((INTENSION
   (PRESENT
    (INTENSION
     (PRED-TO-SET
      (LAMBDA (JX293)
        (LAMBDA (JX294)
          DESTROYER
          (AND
           (EXISTS JX295 CARRIER.VESSEL
            (UNIT.GROUP JX294
             (IOTA JX296 BATTLE.GROUP
              (UNIT.GROUP JX295 JX296))))
           (PRESENT
            (INTENSION
             (UNITS.OVERALL.READINESS JX294 C1)))))
        (IN.PLACE JX293 INDIAN.OCEAN))))))
  TIME WORLD))

(QUERY
 ((INTENSION
   (PRESENT
    (INTENSION
     (PRED-TO-SET
      (LAMBDA (JX288)
        (LAMBDA (JX289)
          DESTROYER
          (EXISTS JX290
           (LAMBDA (JX291)
             CARRIER.VESSEL
             (PRESENT
              (INTENSION
               (UNITS.OVERALL.READINESS JX291 C1))))
           (UNIT.GROUP JX289
            (IOTA JX292 BATTLE.GROUP
             (UNIT.GROUP JX290 JX292)))))
        (IN.PLACE JX288 INDIAN.OCEAN))))))
  TIME WORLD))
```

Figure 4-1: Higher-order forms for distinct readings of the same sentence

the battle group of a C1 carrier is in the Indian Ocean?"

```
(QUERY
 (SET-OF E38
  (AND (AND (DESTROYER E38)
            (AND (EXIST JX295 (CARRIER.VESSEL JX295)
                   (EXIST R101 (UNIT.GROUP R101)
                    (AND (DOMAIN R101 E38)
                         (RANGE R101
                          (IOTA JX296
                           (AND (BATTLE.GROUP JX296)
                                (EXIST R102 (UNIT.GROUP R102)
                                 (AND (DOMAIN R102 JX295)
                                      (RANGE R102 JX296)
                  ))))))))
                  (EXIST 076 (OCCURRENCE 076)
                   (AND (TIMEOFOCCURRENCE 076
                         (ETA T76 (PRESENT T76)))
                        (EXIST R103
                         (UNITS.OVERALL.READINESS R103)
                         (AND (RECORDS 076 R103)
                              (DOMAIN R103 E38)
                              (RANGE R103 C1)))))))
       (EXIST 075 (OCCURRENCE 075)
        (AND (TIMEOFOCCURRENCE 075 (ETA T75 (PRESENT T75)))
             (EXIST R100 (IN.PLACE R100)
              (AND (RECORDS 075 R100)
                   (DOMAIN R100 E38)
                   (RANGE R100 INDIAN.OCEAN)))))))))
```

Figure 4-2: First-order equivalent for the first of the distinct readings

4.2. Answer Generation

We have devised a simple strategy for coupling the higher-order form representing a question with the result returned from the database. Note that in the case of wh- questions like "Which ships are in the Indian Ocean?" we can produce the appropriate logical form for the answer by a simple substitution of the answer set for the question variable, and a change of the speech act operator.

Although true/false questions like "Are there any C4 ships?" will typi-

```
(QUERY
 ((INTENSION
   (PRESENT
    (INTENSION
     (PRED-TO-SET
      (LAMBDA (X) (POWER VESSEL)
       (IN.PLACE X INDIAN.OCEAN))))))
 TIME WORLD))

(ASSERT
 ((INTENSION
   (PRESENT
    (INTENSION
     (IN.PLACE (SET-OF FREDERICK MIDWAY) INDIAN.OCEAN)))))
 TIME WORLD)
```

Figure 4-3: A higher-order question form and its answer form

cally produce a yes/no answer, the availability of the higher-order question form makes it possible to generate a positive answer that carries more information than the minimum necessary for an accurate answer. Thus, if we suppose that a "true" response is indicated by a non-empty answerset from the database, we can easily generate the answer, "Midway is C4." Such an answer has the flavor of being part of a cooperative discourse. For instance, it enables the additional queries "Are there any others?" or "When is it likely to upgrade?" Proper management of English responses in such a discourse is part of our future work.

5. The notion of a transformation

It is often the case that a transformation between formalisms preserves truth values. It is desirable that a form from one representation be true if and only if the transformation of that form into another representation also be true. In our work, we consume higher-order forms by consuming their first-order transformations. The ultimate product is a sentence of English, and as such, we can rely on the disambiguating capacity of the reader to recover the intended sense of expressions that have more than one reading. Consequently, we expect our transformation from logic o language through an intermediate form to preserve truth, in the sense that a dominant reading of the resultant English is equivalent to the original form. This view is silent on the relation of the intermediate form to the original.

A view not silent on that relation is based on the following principles: Any higher-order query that yields a particular answerset given a particular database model will, in its transformed version, yield the same answerset. Also, any assertion in the higher-order language will have the same truth value in its transformed version, again given a particular database model. The axiom schemata we apply are designed to operate in accord with this principle. In doing so, they mediate between a very linguistic ontology, encompassing process occurrences, reified relations, and events, and an ontology parsimonious in these respects for the purpose of reasoning towards the database model. We believe that a special-purpose reasoner, in practice unrealized, would be able to provide equivalent views of the underlying database model.

Our goal, however, is to provide a generation component of general utility, one that will serve in environments other than that of database query. In a more generally useful system with text generation, the considerations that are based on the nature of the underlying database component become irrelevant. An example of such an environment would be an advisory system in which concepts of probability, utility, and obligation would be central to the demands to be expressed. In such an environment, we would have two options. One is to revise the connection of the grammar to its environment in order to accept the meaning representation in higher-order logic, a move we believe possible at the cost of complicating the functions at the interface between the grammar and its environment. The other option is to increase the ability of a first-order language to deal with the added complexity by using a collection of sophisticated axiom schemata to maintain the re-representation of higher-order forms in FOPC. Already we have taken some steps in this direction, again following the tack taken by Hobbs.

6. Summary and Conclusion

We have presented here our system for bridging the gap between a knowledge representation system which uses a higher-order logical language and the Penman generation component. Particular examples have detailed how complex forms involving relational predications and class ascription, as well as operators for tense, abstraction, and set formation, can be reformulated in FOPC.

Additional development is underway regarding the following: (a) the appropriate transformation of operators governing determination and number; (b) the accommodation of the generator's language to facts represented as relations between three or more entities; and (c) the inclusion of discourse demands that lie outside the bounds of the representation language.

Beyond reflecting the system's comprehension by the paraphrase of input sentences, our more long-term future work will be directed toward demonstrating the utility of the generation component as a participant in the process of reporting the results of database query.

Appendix A: Syntax and Semantics for the Higher-Order Intensional Language

We first define the set of types in the following way:[14]

1. TV, INT, T, W, I, and E (for truth value, integer, time, world, individual, and event, respectively), are in TYPE.

2. Whenever ta, tb are in TYPE, then (ta tb) is in TYPE.

3. Whenever ta1,ta2,...,tan are in TYPE, then (TUPLE n ta1,ta2,...,tan) is in TYPE.

4. Whenever ta is in TYPE, then (BAG ta) and (SET ta) are in TYPE.

TYPE-OF is a function which, when given an input x, returns the type of x if x is well-formed, and NIL otherwise. We use the notation := for "returns" or "evaluates to." "(TYPE-OF x) := tx" means "when the TYPE-OF function is applied to the expression x, the value resulting from that application is tx."

We use the convention that parentheses and atoms in capital letters should be taken verbatim, as quoted expressions, but atoms in small letters denote variables and should be evaluated.

Speech acts are in the language, and are used in the interpretation of a sentence. They do not have a type. In this language, the logical form for a sentence will always have the form (<speech-act> ((INTENSION x) time world). The set of speech acts includes QUERY and ASSERT.

1. Variables and constants are represented by atoms.

2. (TYPE-OF (LAMBDA (u) p b)) := (tu tb), if u is a variable of type tu, (TYPE-OF p) := (tu TV), and (TYPE-OF b) := tb.

[14]This abbreviated description of the language was developed by Damaris Ayuso; for a more detailed account, see [Hinrichs et al. 87].

3. (TYPE-OF (f a)) := tb, if (TYPE-OF f) := (ta tb) and (TYPE-OF a) := ta.

4. (TYPE-OF (EQUAL a b)) := TV, if (TYPE-OF a) := ta and (TYPE-OF b) := ta.

5. (TYPE-OF (PRED-TO-SET p)) := (SET ta), if (TYPE-OF p) := (ta TV).

6. (TYPE-OF (CARD s)) := INT, if (TYPE-OF s) := (SET ta).

7. (TYPE-OF (SET-OF a1 a2 ... an)) := (SET ta), if (TYPE-OF a1), (TYPE-OF a2),..., and (TYPE-OF an) := ta.

8. (TYPE-OF (FORALL u p b)) := TV, and (TYPE-OF (EXISTS u p b)) := TV, if u is a variable of type tu, (TYPE-OF p) := (tu TV), and (TYPE-OF b) := TV.

9. (TYPE-OF (IOTA u p b)) := tu, if u is a variable of type tu, (TYPE-OF p) := (tu TV), and (TYPE-OF b) := TV.

10. (TYPE-OF (G i)) := (I TV), if (TYPE-OF i) := (W (T (I TV))).

11. (TYPE-OF (SAMPLE i)) := (I TV), if (TYPE-OF i) := (W (T (I TV))).

12. (TYPE-OF (KIND-OF i)) := I, if (TYPE-OF i) := (W (T (I TV))).

13. (TYPE-OF (TUPLE-OF n a1 a2 ... an)) := (TUPLE n ta1 ta2 ... tan), if (TYPE-OF a1) := ta1, (TYPE-OF a2) := ta2, ..., (TYPE-OF an) := tan.

14. (TYPE-OF (UNION p)) := (ta TV), if (TYPE-OF p) := ((ta TV) TV).

15. (TYPE-OF (POWER p)) := ((ta TV) TV), if (TYPE-OF p) := (ta TV).

16. (TYPE-OF (APPLY-COLLECT f s)) := (BAG tb), if (TYPE-OF s) := (SET ta) or (TYPE-OF s) := (BAG ta), and (TYPE-OF f) := (ta tb).

17. (TYPE-OF (MEMBER a s)) := TV, if (TYPE-OF s) := (SET ta) and (TYPE-OF a) := ta.

18. (TYPE-OF (BAG-TO-SET b)) := (SET tb), if (TYPE-OF b) := (BAG tb).

19. (TYPE-OF (AND f1 f2)), (TYPE-OF (OR f1 f2)), (TYPE-OF (IMPLIES f1 f2)), (TYPE-OF (IFF f1 f20)), (TYPE-OF (NOT f1)), (TYPE-OF (NECESSARILY f1)), and (TYPE-OF (POSSIBLY f1)) := TV, if (TYPE-OF f1) := TV and (TYPE-OF f2) := TV.

20. (TYPE-OF (INTENSION a)) := (W (T ta)), if (TYPE-OF a) := ta.

21. (TYPE-OF (EXTENSION i)) := ta, if (TYPE-OF i) := (W (T ta)).

22. (TYPE-OF (<tense-operator> i))) := ta, where <tense-operator> is one of PRESENT, PAST, FUTURE, PRESPERF, and PASTPERF, if (TYPE-OF i) := (W (T ta)).

23. Nothing is in any set ME-ta, except as specified in the above clauses.[15]

[15]As this specification stands, the following problem arises. Consider the expression

```
(PRED-TO-SET (LAMBDA (X) VESSEL (HARP.CAPABLE.SHIP X))).
```

We expect the type of this expression to be (SET I), that is, a set of individuals. Consequently, the type of the LAMBDA expression should be of type (I TV)--functions from individuals to truth values. Accordingly, the type of X should be I, that of VESSEL (I TV), and that of (HARP.CAPABLE.SHIP X) TV. This is as it should be. Now, however, consider the expression

```
(PRED-TO-SET (LAMBDA (X) (POWER VESSEL) (IN.PLACE X INDIAN.OCEAN))).
```

The type of (POWER VESSEL) is ((I TV) TV). The type of (IN.PLACE X INDIAN.OCEAN) is TV. (We assume a Currying operation is applied to bring this expression into accord with the specification, which allows only one-place arguments.) The type of the LAMBDA expression is then ((I TV) TV) and the type of the whole expression is (SET (I TV)). But the nature of the English query, "Which ships are in the Indian Ocean?", the meaning representation for which includes this formula as a component, would lead one to expect a different result--a set of individual ships. This reflects not so much a problem in the specification as a problem in the decision to represent plurals with the power set notation.

Appendix B: BNF description of the first-order MRL

MRL Statements

MRLStatement := (**ASSERT** *MRLFormula*) |
(**QUERY** (*MRLFormula* | *MRLTerm*)) |
(**COMMAND** *MRLTerm*) |
(**ANSWER** (**YES** | **NO**))

MRL Terms

MRLTerm := *Constant* | *Variable* |
(**SET-OF** *Variable MRLFormula*) |
(**SET-WITH** (*Constant*)$^+$) |
(**IOTA** *Variable MRLFormula*) |
(**ETA** *Variable MRLFormula*)
Constant := *LispAtom*
Variable := *LispAtom*

MRL Formulas

MRLFormula := *MRLAtomicFormula* |
((**AND** | **OR**) *MRLFormula MRLFormula*$^+$) |
((**NOT** *MRLFormula*) |
(*Quantifier Variable Range Predication*)
Quantifier := **ALL** | **EXIST** | *ExtendedQuantifier*
ExtendedQuantifier := **EXIST1!** | **EXIST1?** | **EXIST>1?**
Range := *MRLFormula*
Predication := *MRLFormula*
MRLAtomicFormula := (*UnaryPredicateMRLTerm*) |
(*BinaryPredicateMRLTermMRLTerm*)
UnaryPredicate := *NIKLConceptName*
BinaryPredicate := *NIKLRoleName*

The **SET-OF** operator generates a set according to the description of *MRLFormula*, which should be an open formula with respect to the *Variable*. Actually, an explicit set is only required in the context of a **QUERY** Statement or when an assertion concerning cardinality is made. The **SET-WITH** operator specifies a set by enumerating the elements.

IOTA and **ETA** are both operators determining one domain constant. The same well-formedness conditions as for sets apply here. IOTA determines the one element described by the formula, i.e. IOTA carries the presupposition that there is exactly one element which fits the description. The ETA operator in contrast to that is intended to refer to an object which is not fully described, but which definitely exists (perhaps in the future), and which can be referred to; i.e. ETA captures one meaning of the indefinite singular determiner.

In addition to the standard quantifiers **ALL** and **EXIST**, some extended quantifiers are used. The following meaning is intended (assuming that (P x), (Q x), and (R x) are open formulas with respect to x):

$$
\begin{aligned}
&(\textbf{EXIST}\, x\, (R\, x)(P x)) \equiv (\exists\, x)((R\, x) \wedge (P x)) \\
&(\textbf{ALL}\, x\, (R\, x)(P x)) \equiv (\forall\, x)((R\, x) \rightarrow (P x)) \\
&(\textbf{EXIST1!}\, x\, (R\, x)(P x)) \equiv (P(\iota x)(R\, x)) \\
&\qquad\qquad \equiv (\exists\, x)((R\, x) \wedge (P x) \wedge (\forall\, y)((R\, y) \rightarrow x{=}y)) \\
&(\textbf{EXIST1?}\, x\, (R\, x)(P x)) \equiv (P(\eta x)(R\, x)) \\
&\qquad\qquad \equiv (\exists\, x)((R\, x) \wedge (Q\, x) \wedge (P x) \wedge (\forall\, y)((R\, y) \wedge (Q\, y) \rightarrow x{=}y)) \\
&(\textbf{EXIST}{>}\textbf{1?}\, x\, (R\, x)(P x)) \equiv \\
&\qquad\qquad (\exists\, s)(s \subseteq \{y \mid (R\, y)\} \wedge |s| > 1 \wedge (\forall\, x)(x \in s \rightarrow (P x)))
\end{aligned}
$$

The extended quantifiers do not provide a higher degree of expressibility than standard quantifiers. Except for EXIST>1?, which is intended to capture the 'plural' ETA case, they provide another way of expressing ETA and IOTA with the advantage that the bound variable can be used more than once in one formula.

Appendix C: Program design and execution

The function WML-PARSE takes a well-formed expression from the higher-order language, represented as an s-expression, and parses it into a data structure where each component is represented as a structure of the appropriate type, according to the syntax and semantics specification for the language. An ill-formed expression will cause an error condition to be signalled.

A synthesis function may be applied to the resulting structure and compared against the original input to verify the accuracy of the parse.

The structure output from WML-PARSE becomes the input to the function WML-TRANSFORM, which represents the transformation for each type of WML sub-formula. Each case produces the appropriate structure for an MRL expression. This MRL structure is then passed to the function MRL-SYN which constructs an s-expression representing the derived MRL formula.

The following exhibits the operation of these steps for several examples.

"Which ships are in the Indian Ocean?"

Higher-order form:

```
(QUERY
 ((INTENSION
   (PRESENT
    (INTENSION
     (PRED-TO-SET
      (LAMBDA (JX145) (POWER VESSEL)
       (IN.PLACE JX145 INDIAN.OCEAN))))))
  TIME WORLD))
```

Structure representing parse of higher-order form:

```
#S(SPEECHACT
  :TYPE QUERY
  :INTENSION
   #S(INTENSION
    :FORM
     #S(TENSE
       :TYPE PRESENT
       :INTENSION
        #S(INTENSION
          :FORM
           #S(PRED-TO-SET
             :INTENSION
              #S(LAMBDA
                :VAR #S(IND :SYMBOL JX145)
                :RANGE #S(POWER :PRED #S(PRED :SYMBOL VESSEL))
                :PRED
                #S(GROUND-ROLE
                 :PREDICATE #S(PRED :SYMBOL IN.PLACE)
                 :DOMAIN #S(IND :SYMBOL JX145)
                 :RANGE #S(IND :SYMBOL INDIAN.OCEAN))))))
 :TIME TIME
 :WORLD WORLD)
```

MRL structure representing transformation:

```
#S(MRL-STATEMENT
 :TYPE QUERY
 :FORM
  #S(MRL-SET-OF-TERM
   :VAR #S(MRL-IND :SYMBOL E20)
   :FORM
    #S(MRL-CONJUNCTION
     :COMPONENTS
      (#S(MRL-UNARY-PREDICATE
         :PRED VESSEL
         :TERM #S(MRL-IND :SYMBOL E20))
        #S(MRL-QUANTIFIED-FORMULA
           :QUANTIFIER EXIST1!
           :VAR #S(MRL-IND :SYMBOL O51)
           :RANGE
            #S(MRL-UNARY-PREDICATE
             :PRED OCCURRENCE
             :TERM #S(MRL-IND :SYMBOL O51))
           :PRED
            #S(MRL-QUANTIFIED-FORMULA
             :QUANTIFIER EXIST1!
             :VAR #S(MRL-IND :SYMBOL R98)
             :RANGE
              #S(MRL-UNARY-PREDICATE
               :PRED IN.PLACE
               :TERM #S(MRL-IND :SYMBOL R98))
             :PRED
              #S(MRL-CONJUNCTION
               :COMPONENTS
                (#S(MRL-BINARY-PREDICATE
                  :PRED TIMEOFOCCURRENCE
                  :FIRST #S(MRL-IND :SYMBOL O51)
                  :SECOND
                   #S(MRL-IND-TERM
                    :TYPE ETA
                    :VAR #S(MRL-IND :SYMBOL T51)
                    :FORM
```

```
                                    #S(MRL-UNARY-PREDICATE
                                      :PRED PRESENT
                                      :TERM #S(MRL-IND :SYMBOL T51)))
                                #S(MRL-BINARY-PREDICATE
                                  :PRED RECORDS
                                  :FIRST #S(MRL-IND :SYMBOL 051)
                                  :SECOND #S(MRL-IND :SYMBOL R98))
                                #S(MRL-BINARY-PREDICATE
                                  :PRED DOMAIN
                                  :FIRST #S(MRL-IND :SYMBOL R98)
                                  :SECOND #S(MRL-IND :SYMBOL E20))
                                #S(MRL-BINARY-PREDICATE
                                  :PRED RANGE
                                  :FIRST #S(MRL-IND :SYMBOL R98)
                                  :SECOND
                                  #S(MRL-IND :SYMBOL INDIAN.OCEAN)))))))))
```

Resulting first-order expression:

```
(QUERY
 (SET-OF E20
  (AND (VESSEL E20)
       (EXIST1! 051 (OCCURRENCE 051)
        (EXIST1! R98 (IN.PLACE R98)
         (AND (TIMEOFOCCURRENCE 051 (ETA T51 (PRESENT T51)))
              (RECORDS 051 R98)
              (DOMAIN R98 E20)
              (RANGE R98 INDIAN.OCEAN)))))))
```

"Are any US ships in the Indian Ocean?"

Higher-order form:

```
(QUERY
 ((INTENSION
   (PRESENT
    (INTENSION
     (EXISTS JX146
      (LAMBDA (JX136) VESSEL (HOME.COUNTRY JX136 USA))
      (IN.PLACE JX146 INDIAN.OCEAN)))))
  TIME WORLD))
```

Structure representing parse of higher-order form:

```
#S(SPEECHACT
  :TYPE QUERY
  :INTENSION
```

```
 #S(INTENSION
  :FORM
   #S(TENSE
    :TYPE PRESENT
    :INTENSION
     #S(INTENSION
      :FORM
       #S(QUANTIFIED-FORM
        :QUANTIFIER EXISTS
        :VAR #S(IND :SYMBOL JX146)
        :RANGE
         #S(LAMBDA
          :VAR #S(IND :SYMBOL JX136)
          :RANGE #S(PRED :SYMBOL VESSEL)
          :PRED
           #S(GROUND-ROLE
            :PREDICATE #S(PRED :SYMBOL HOME.COUNTRY)
            :DOMAIN #S(IND :SYMBOL JX136)
            :RANGE #S(IND :SYMBOL USA)))
        :PRED
         #S(GROUND-ROLE
          :PREDICATE #S(PRED :SYMBOL IN.PLACE)
          :DOMAIN #S(IND :SYMBOL JX146)
          :RANGE #S(IND :SYMBOL INDIAN.OCEAN))))))
 :TIME TIME
 :WORLD WORLD)
```

MRL structure representing transformation:

```
#S(MRL-STATEMENT
 :TYPE QUERY
 :FORM
  #S(MRL-QUANTIFIED-FORMULA
   :QUANTIFIER EXIST
   :VAR #S(MRL-IND :SYMBOL JX146)
   :RANGE
    #S(MRL-CONJUNCTION
     :COMPONENTS
      (#S(MRL-UNARY-PREDICATE
        :PRED VESSEL
        :TERM #S(MRL-IND :SYMBOL JX146))
       #S(MRL-QUANTIFIED-FORMULA
        :QUANTIFIER EXIST1!
        :VAR #S(MRL-IND :SYMBOL R100)
        :RANGE
 #S(MRL-UNARY-PREDICATE
   :PRED HOME.COUNTRY
   :TERM #S(MRL-IND :SYMBOL R100))
   :PRED
    #S(MRL-CONJUNCTION
      :COMPONENTS
       (#S(MRL-BINARY-PREDICATE
          :PRED DOMAIN
          :FIRST #S(MRL-IND :SYMBOL R100)
          :SECOND #S(MRL-IND :SYMBOL JX146))
    #S(MRL-BINARY-PREDICATE
      :PRED RANGE
      :FIRST #S(MRL-IND :SYMBOL R100)
      :SECOND #S(MRL-IND :SYMBOL USA))))))
:PRED
 #S(MRL-QUANTIFIED-FORMULA
   :QUANTIFIER EXIST1!
   :VAR #S(MRL-IND :SYMBOL 052)
   :RANGE
    #S(MRL-UNARY-PREDICATE
      :PRED OCCURRENCE
      :TERM #S(MRL-IND :SYMBOL 052))
      :PRED
       #S(MRL-QUANTIFIED-FORMULA
         :QUANTIFIER EXIST1!
         :VAR #S(MRL-IND :SYMBOL R99)
         :RANGE
          #S(MRL-UNARY-PREDICATE
            :PRED IN.PLACE
            :TERM #S(MRL-IND :SYMBOL R99))
         :PRED
          #S(MRL-CONJUNCTION
            :COMPONENTS
             (#S(MRL-BINARY-PREDICATE
                :PRED TIMEOFOCCURRENCE
                :FIRST #S(MRL-IND :SYMBOL 052)
                :SECOND
                 #S(MRL-IND-TERM
                   :TYPE ETA
                   :VAR #S(MRL-IND :SYMBOL T52)
                   :FORM
                    #S(MRL-UNARY-PREDICATE
                      :PRED PRESENT
                      :TERM #S(MRL-IND :SYMBOL T52))))
                    #S(MRL-BINARY-PREDICATE
                     :PRED RECORDS
                     :FIRST #S(MRL-IND :SYMBOL O52)
                     :SECOND #S(MRL-IND :SYMBOL R99))
                    #S(MRL-BINARY-PREDICATE
                     :PRED DOMAIN
                     :FIRST #S(MRL-IND :SYMBOL R99)
                     :SECOND #S(MRL-IND :SYMBOL JX146))
                    #S(MRL-BINARY-PREDICATE
                     :PRED RANGE
                     :FIRST #S(MRL-IND :SYMBOL R99)
                     :SECOND #S(MRL-IND
                                :SYMBOL INDIAN.OCEAN)))))))
```

Resulting first-order expression:

```
(QUERY
 (EXIST JX146
  (AND (VESSEL JX146)
       (EXIST1! R100 (HOME.COUNTRY R100)
        (AND (DOMAIN R100 JX146)
             (RANGE R100 USA))))
  (EXIST1! O52 (OCCURRENCE O52)
   (EXIST1! R99 (IN.PLACE R99)
    (AND (TIMEOFOCCURRENCE O52 (ETA T52 (PRESENT T52)))
         (RECORDS O52 R99)
         (DOMAIN R99 JX146)
         (RANGE R99 INDIAN.OCEAN))))))
```

"Has every ship in Frederick's battle group been deployed?"

Higher-order form:

```
(QUERY
 ((INTENSION
   (PRESPERF
    (INTENSION
     (FORALL JX113
      (LAMBDA (JX107)
        VESSEL
        (UNIT.GROUP JX107
         (IOTA JX108 BATTLE.GROUP
          (UNIT.GROUP
           (IOTA JX109 VESSEL (NAMEOF JX109 FREDERICK))
           JX108))))
      (EXISTS JX114 DEPLOYMENT (OBJECT.OF JX114 JX113))))))
   TIME WORLD))
```

Structure representing parse of higher-order form:

```
#S(SPEECHACT
 :TYPE QUERY
 :INTENSION
  #S(INTENSION
   :FORM
    #S(TENSE
     :TYPE PRESPERF
     :INTENSION
      #S(INTENSION
       :FORM
        #S(QUANTIFIED-FORM
         :QUANTIFIER FORALL
         :VAR #S(IND :SYMBOL JX113)
         :RANGE
          #S(LAMBDA
           :VAR #S(IND :SYMBOL JX107)
           :RANGE #S(PRED :SYMBOL VESSEL)
           :PRED
            #S(GROUND-ROLE
             :PREDICATE #S(PRED :SYMBOL UNIT.GROUP)
             :DOMAIN #S(IND :SYMBOL JX107)
             :RANGE
              #S(IOTA
               :VAR #S(IND :SYMBOL JX108)
               :RANGE #S(PRED :SYMBOL BATTLE.GROUP)
               :PRED
                #S(GROUND-ROLE
                 :PREDICATE #S(PRED :SYMBOL UNIT.GROUP)
                 :DOMAIN
                  #S(IOTA :VAR #S(IND :SYMBOL JX109)
                   :RANGE #S(PRED :SYMBOL VESSEL)
                   :PRED
                    #S(GROUND-ROLE
                     :PREDICATE #S(PRED :SYMBOL NAMEOF)
                      :DOMAIN #S(IND :SYMBOL JX109)
                      :RANGE #S(IND :SYMBOL FREDERICK)))
                 :RANGE #S(IND :SYMBOL JX108)))))
         :PRED
          #S(QUANTIFIED-FORM
           :QUANTIFIER EXISTS
           :VAR #S(IND :SYMBOL JX114)
```

```
              :RANGE #S(PRED :SYMBOL DEPLOYMENT)
              :PRED
               #S(GROUND-ROLE
                :PREDICATE #S(PRED :SYMBOL OBJECT.OF)
                :DOMAIN #S(IND :SYMBOL JX114)
                :RANGE #S(IND :SYMBOL JX113))))))
 :TIME TIME
 :WORLD WORLD)
```

MRL structure representing transformation:

```
#S(MRL-STATEMENT
 :TYPE QUERY
 :FORM
  #S(MRL-QUANTIFIED-FORMULA
   :QUANTIFIER ALL
   :VAR #S(MRL-IND :SYMBOL JX113)
   :RANGE
    #S(MRL-CONJUNCTION
     :COMPONENTS
      (#S(MRL-UNARY-PREDICATE
        :PRED VESSEL
        :TERM #S(MRL-IND :SYMBOL JX113))
       #S(MRL-QUANTIFIED-FORMULA
        :QUANTIFIER EXIST1!
        :VAR #S(MRL-IND :SYMBOL R101)
        :RANGE
         #S(MRL-UNARY-PREDICATE
          :PRED UNIT.GROUP
          :TERM #S(MRL-IND :SYMBOL R101))
          :PRED
           #S(MRL-CONJUNCTION
            :COMPONENTS
             (#S(MRL-BINARY-PREDICATE
               :PRED DOMAIN
               :FIRST #S(MRL-IND :SYMBOL R101)
               :SECOND #S(MRL-IND :SYMBOL JX113))
              #S(MRL-BINARY-PREDICATE
               :PRED RANGE
               :FIRST #S(MRL-IND :SYMBOL R101)
               :SECOND
                #S(MRL-IND-TERM
                 :TYPE IOTA
                 :VAR #S(MRL-IND :SYMBOL JX108)
                 :FORM
```

```
#S(MRL-CONJUNCTION
 :COMPONENTS
  (#S(MRL-UNARY-PREDICATE
    :PRED BATTLE.GROUP
    :TERM #S(MRL-IND :SYMBOL JX108))
   #S(MRL-QUANTIFIED-FORMULA
    :QUANTIFIER EXIST1!
    :VAR #S(MRL-IND :SYMBOL R102)
    :RANGE
     #S(MRL-UNARY-PREDICATE
      :PRED UNIT.GROUP
      :TERM #S(MRL-IND :SYMBOL R102))
    :PRED
     #S(MRL-CONJUNCTION
      :COMPONENTS
       (#S(MRL-BINARY-PREDICATE
         :PRED DOMAIN
         :FIRST #S(MRL-IND :SYMBOL R102)
         :SECOND
          #S(MRL-IND-TERM
           :TYPE IOTA
           :VAR #S(MRL-IND :SYMBOL JX109)
           :FORM
            #S(MRL-CONJUNCTION
             :COMPONENTS
              (#S(MRL-UNARY-PREDICATE
                :PRED VESSEL
                :TERM
                 #S(MRL-IND
                    :SYMBOL JX109))
               #S(MRL-QUANTIFIED-FORMULA
                :QUANTIFIER EXIST1!
                :VAR #S(MRL-IND
                        :SYMBOL R103)
                :RANGE
                 #S(MRL-UNARY-PREDICATE
                  :PRED NAMEOF
                  :TERM #S(MRL-IND
                           :SYMBOL R103))
                :PRED
                 #S(MRL-CONJUNCTION
                  :COMPONENTS
                  (#S(MRL-BINARY-PREDICATE
                     :PRED DOMAIN
```

```
                                                  :FIRST
                                                   #S(MRL-IND
                                                      :SYMBOL R103)
                                                  :SECOND
                                                   #S(MRL-IND
                                                      :SYMBOL JX109))
                                               #S(MRL-BINARY-PREDICATE
                                                  :PRED RANGE
                                                  :FIRST
                                                   #S(MRL-IND
                                                      :SYMBOL R103)
                                                  :SECOND
                                                   #S(MRL-IND
                                                     :SYMBOL FREDERICK)
                                                     ))))))))
                               #S(MRL-BINARY-PREDICATE
                                :PRED RANGE
                                :FIRST #S(MRL-IND :SYMBOL R102)
                                :SECOND
                                #S(MRL-IND :SYMBOL JX108))))))))))))
                   :PRED
                    #S(MRL-QUANTIFIED-FORMULA
                     :QUANTIFIER EXIST1!
                     :VAR #S(MRL-IND :SYMBOL 053)
                     :RANGE
                      #S(MRL-UNARY-PREDICATE
                      :PRED OCCURRENCE
                      :TERM #S(MRL-IND :SYMBOL 053))
                     :PRED
                      #S(MRL-QUANTIFIED-FORMULA
                       :QUANTIFIER EXIST
                       :VAR #S(MRL-IND :SYMBOL JX114)
                       :RANGE
                        #S(MRL-UNARY-PREDICATE
                         :PRED DEPLOYMENT
                         :TERM #S(MRL-IND :SYMBOL JX114))
                       :PRED
                        #S(MRL-CONJUNCTION
                         :COMPONENTS
                          (#S(MRL-BINARY-PREDICATE
                              :PRED TIMEOFOCCURRENCE
                              :FIRST #S(MRL-IND :SYMBOL 053)
                              :SECOND
                               #S(MRL-IND-TERM
                        :TYPE ETA
                        :VAR #S(MRL-IND :SYMBOL T53)
                        :FORM
                         #S(MRL-UNARY-PREDICATE
                           :PRED PRESPERF
                           :TERM #S(MRL-IND :SYMBOL T53)))
                   #S(MRL-BINARY-PREDICATE
                     :PRED RECORDS
                     :FIRST #S(MRL-IND :SYMBOL 053)
                     :SECOND #S(MRL-IND :SYMBOL JX114))
                   #S(MRL-BINARY-PREDICATE
                     :PRED OBJECT.OF
                     :FIRST #S(MRL-IND :SYMBOL JX114)
                     :SECOND #S(MRL-IND :SYMBOL JX113)))))))
```

Resulting first-order expression:

```
(QUERY
 (ALL JX113
  (AND (VESSEL JX113)
       (EXIST1! R101 (UNIT.GROUP R101)
        (AND (DOMAIN R101 JX113)
             (RANGE R101
              (IOTA JX108
               (AND (BATTLE.GROUP JX108)
                    (EXIST1! R102 (UNIT.GROUP R102)
                     (AND (DOMAIN R102
                           (IOTA JX109
                            (AND (VESSEL JX109)
                                 (EXIST1! R103 (NAMEOF R103)
                                  (AND (DOMAIN R103 JX109)
                                       (RANGE R103 FREDERICK)
                          )))))
                          (RANGE R102 JX108))))))))
  (EXIST1! 053 (OCCURRENCE 053)
   (EXIST JX114 (DEPLOYMENT JX114)
    (AND (TIMEOFOCCURRENCE 053 (ETA T53 (PRESPERF T53)))
         (RECORDS 053 JX114)
         (OBJECT.OF JX114 JX113))))))
```

"What ships that are deployed are harpoon capable?"

Higher-order form:

```
(QUERY
 ((INTENSION
 (PRESENT
  (INTENSION
   (PRED-TO-SET
    (LAMBDA (JX122)
     (POWER
      (LAMBDA (JX123) VESSEL
       (PRESENT
        (INTENSION
         (EXISTS JX124 DEPLOYMENT
          (OBJECT.OF JX124 JX123))))))
     (HARP.CAPABLE.SHIP JX122))))))
TIME WORLD))
```

Structure representing parse of higher-order form:

```
#S(SPEECHACT
 :TYPE QUERY
 :INTENSION
  #S(INTENSION
   :FORM
    #S(TENSE
     :TYPE PRESENT
     :INTENSION
      #S(INTENSION
       :FORM
        #S(PRED-TO-SET
         :INTENSION
          #S(LAMBDA
           :VAR #S(IND :SYMBOL JX122)
           :RANGE
            #S(POWER
             :PRED
              #S(LAMBDA
               :VAR #S(IND :SYMBOL JX123)
               :RANGE #S(PRED :SYMBOL VESSEL)
               :PRED
                #S(TENSE
                 :TYPE PRESENT
                 :INTENSION
                  #S(INTENSION
                   :FORM
                    #S(QUANTIFIED-FORM
                     :QUANTIFIER EXISTS
                     :VAR #S(IND :SYMBOL JX124)
                     :RANGE #S(PRED :SYMBOL DEPLOYMENT)
```

```
                              :PRED
                               #S(GROUND-ROLE
                                 :PREDICATE #S(PRED :SYMBOL OBJECT.OF)
                                 :DOMAIN #S(IND :SYMBOL JX124)
                                 :RANGE #S(IND :SYMBOL JX123)))))))
                    :PRED
                     #S(GROUND-CONCEPT
                       :PREDICATE #S(PRED :SYMBOL HARP.CAPABLE.SHIP)
                       :IND #S(IND :SYMBOL JX122)))))))
  :TIME TIME
  :WORLD WORLD)
```

MRL structure representing transformation:

```
#S(MRL-STATEMENT
  :TYPE QUERY
  :FORM
   #S(MRL-SET-OF-TERM
     :VAR #S(MRL-IND :SYMBOL E23)
     :FORM
      #S(MRL-CONJUNCTION
        :COMPONENTS
         (#S(MRL-CONJUNCTION
            :COMPONENTS
             (#S(MRL-UNARY-PREDICATE
                :PRED VESSEL
                :TERM #S(MRL-IND :SYMBOL E23))
              #S(MRL-QUANTIFIED-FORMULA
                :QUANTIFIER EXIST1!
                :VAR #S(MRL-IND :SYMBOL 055)
                :RANGE
                 #S(MRL-UNARY-PREDICATE
                   :PRED OCCURRENCE
                   :TERM #S(MRL-IND :SYMBOL 055))
                :PRED
                 #S(MRL-QUANTIFIED-FORMULA
                   :QUANTIFIER EXIST
                   :VAR #S(MRL-IND :SYMBOL JX124)
                   :RANGE
                    #S(MRL-UNARY-PREDICATE
                      :PRED DEPLOYMENT
                      :TERM #S(MRL-IND :SYMBOL JX124))
                   :PRED
                    #S(MRL-CONJUNCTION
                      :COMPONENTS
```

```
           (#S(MRL-BINARY-PREDICATE
              :PRED TIMEOFOCCURRENCE
              :FIRST #S(MRL-IND :SYMBOL O55)
              :SECOND
               #S(MRL-IND-TERM
                 :TYPE ETA
                 :VAR #S(MRL-IND :SYMBOL T55)
                 :FORM
                  #S(MRL-UNARY-PREDICATE
                    :PRED PRESENT
                    :TERM #S(MRL-IND :SYMBOL T55))))
            #S(MRL-BINARY-PREDICATE
              :PRED RECORDS
              :FIRST #S(MRL-IND :SYMBOL O55)
              :SECOND #S(MRL-IND :SYMBOL JX124))
            #S(MRL-BINARY-PREDICATE
              :PRED OBJECT.OF
              :FIRST #S(MRL-IND :SYMBOL JX124)
              :SECOND #S(MRL-IND :SYMBOL E23)))))))
#S(MRL-QUANTIFIED-FORMULA
  :QUANTIFIER EXIST1!
  :VAR #S(MRL-IND :SYMBOL O54)
  :RANGE
   #S(MRL-UNARY-PREDICATE
     :PRED OCCURRENCE
     :TERM #S(MRL-IND :SYMBOL O54))
  :PRED
   #S(MRL-QUANTIFIED-FORMULA
     :QUANTIFIER EXIST1!
     :VAR #S(MRL-IND :SYMBOL R104)
     :RANGE
      #S(MRL-UNARY-PREDICATE
        :PRED CLASSASCRIPTION
        :TERM #S(MRL-IND :SYMBOL R104))
     :PRED
      #S(MRL-CONJUNCTION
        :COMPONENTS
         (#S(MRL-BINARY-PREDICATE
            :PRED TIMEOFOCCURRENCE
            :FIRST #S(MRL-IND :SYMBOL O54)
            :SECOND
             #S(MRL-IND-TERM
               :TYPE ETA
               :VAR #S(MRL-IND :SYMBOL T54)
```

```
                    :FORM
                     #S(MRL-UNARY-PREDICATE
                       :PRED PRESENT
                       :TERM #S(MRL-IND :SYMBOL T54)))
                #S(MRL-BINARY-PREDICATE
                  :PRED RECORDS
                  :FIRST #S(MRL-IND :SYMBOL O54)
                  :SECOND #S(MRL-IND :SYMBOL R104))
                #S(MRL-BINARY-PREDICATE
                  :PRED DOMAIN
                  :FIRST #S(MRL-IND :SYMBOL R104)
                  :SECOND #S(MRL-IND :SYMBOL E23))
                #S(MRL-BINARY-PREDICATE
                  :PRED RANGE
                  :FIRST #S(MRL-IND :SYMBOL R104)
                  :SECOND (QUOTE HARP.CAPABLE.SHIP)))))))))
```

Resulting first-order expression[16]:

```
(QUERY
 (SET-OF E23
  (AND (AND (VESSEL E23)
            (EXIST1! O55 (OCCURRENCE O55)
             (EXIST JX124 (DEPLOYMENT JX124)
              (AND (TIMEOFOCCURRENCE O55
                    (ETA T55 (PRESENT T55)))
                   (RECORDS O55 JX124)
                   (OBJECT.OF JX124 E23)))))
       (EXIST1! O54 (OCCURRENCE O54)
        (EXIST1! R104 (CLASSASCRIPTION R104)
         (AND (TIMEOFOCCURRENCE O54 (ETA T54 (PRESENT T54)))
              (RECORDS O54 R104)
              (DOMAIN R104 E23)
              (RANGE R104 'HARP.CAPABLE.SHIP)))))))
```

[16]In this form we use the notation 'HARP.CAPABLE.SHIP as an abbreviation for the expression (SET-OF X (HARP.CAPABLE.SHIP X)), i.e. to represent the intensional construction of the set by reference to the concept on which it is based.

Appendix D: Code

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;COPYRIGHT (C) UNIVERSITY OF SOUTHERN CALIFORNIA, 1987      ;
;University Of Southern California, Information Sciences    ;
;Institute, 4676 Admiralty Way, Marina Del Rey, California ;
;                                                           ;
;This software was developed under the terms and conditions;
;of Contract No. MDA903 81 C 0335 between the Defense       ;
;Advanced Research Projects Agency and the University of    ;
;Southern California, Information Sciences Institute. Use   ;
;and distribution of this software is further subject to    ;
;the provisions of that contract and any other agreements   ;
;developed between the user of the software and the         ;
;University of Southern California, Information Sciences    ;
;Institute.  This software is within the scope of the       ;
;"AGREEMENT FOR SOFTWARE EXCHANGE" between BBN Laboratories;
;Incorporated and the University of Southern California,    ;
;dated February 18, 1986.  It is supplied "AS IS," without ;
;any warranties of any kind.  It is furnished only on the   ;
;basis that any party who receives it indemnifies and holds;
;harmless the parties who furnish and originate it against ;
;any claims, demands, or liabilities connected with using   ;
;it, furnishing it to others or providing it to a third     ;
;party.  THIS NOTICE MUST NOT BE REMOVED FROM THE SOFTWARE,;
;AND IN THE EVENT THAT THE SOFTWARE IS DIVIDED, IT SHOULD   ;
;BE ATTACHED TO EVERY PART.                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

WML-STRUCTURES.LISP
;; -*- Syntax: Common-lisp; Package: PENMAN; Mode: LISP; -*-

(in-package 'penman)

(defconstant wml-ops '(query assert lambda intension equal
                         pred-to-set card set-of forall
                         exists iota sample power
                         and not time world present past
                         future presperf pastperf))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  Structures for parsing BBN WML, based on observed forms.;
;  Not truly general with regard to types:  EQUAL here      ;
;  relates only INDs, but the spec allows things of         ;
;  identical type.                                          ;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defstruct card
  "WML SPEC #6"
  (set nil :type set))

(defstruct conjunct
  "WML SPEC #19"
  (components nil :type list))

(defstruct equal
  "WML SPEC #4"
  (first  nil :type i)
  (second nil :type i))

(defstruct ground-concept
  "WML SPEC #3"
  (predicate nil :type pred)
  (ind       nil :type i   ))

(defstruct ground-role
  "WML SPEC #3"
  (predicate   nil :type pred)
  (domain      nil :type i)
  (range       nil :type i))

(defstruct ind
  "WML SPEC #1"
  (symbol nil :type atom))

(defstruct intension
  "WML SPEC #20"
  (form nil :type (or ind-to-tv tv)))

(defstruct iota
  "WML SPEC #9"
  (var    nil :type ind      )
  (range  nil :type ind-to-tv)
  (pred   nil :type tv       ))

(defstruct lambda
  "WML SPEC #2"
  (var    nil :type ind      )
  (range  nil :type ind-to-tv)
  (pred   nil :type tv       ))

(defstruct negation
  "WML SPEC #19"
  (form nil :type tv))

(defstruct power
  "WML SPEC #15"
  (pred nil :type tv))

(defstruct pred
  "WML SPEC #3"
  (symbol nil :type atom))

(defstruct pred-to-set
  "WML SPEC #5"
  (intension nil :type tv))

(defstruct quantified-form
  "WML SPEC #8"
  (quantifier  nil :type atom     )
  (var         nil :type ind      )
  (range       nil :type ind-to-tv)
  (pred        nil :type tv       ))

(defstruct sample
  "WML SPEC #11"
  (intension nil :type intension))

(defstruct set-of
  "WML SPEC #7"
  (extension nil :type list))

(defstruct speechact
  "WML SPEC #0"
  (type       nil    :type atom     )
  (intension  nil    :type intension)
  (time       'time  :type atom     )
  (world      'world :type atom     ))

(defstruct tense
  "WML SPEC #22"
  (type      nil :type atom     )
  (intension nil :type intension))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Composed types to specify complex structures in BBN WML   ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(deftype i ()
  '(or ind iota))

(deftype ind-to-tv ()
  '(or pred
       lambda
       power
       sample))

(deftype set ()
  '(or pred-to-set set-of))

(deftype tv ()
  '(or conjunct
       equal
       ground-concept
       ground-role
       negation
       quantified-form))

(deftype wml-wff ()
  '(or i
       ind-to-tv
       set
       tv
       speechact))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Predicates for identifying complex structures in BBN WML ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun i-p (form)
  (or (and (atom form)
           (not (member form wml-ops)))
      (iota-p form)))

(defun ind-to-tv-p (form)
  (or (and (atom form)
           (not (member form wml-ops)))
      (lambda-p form)
      (power-p  form)
      (sample-p form)))

(defun set-p (form)
  (or (pred-to-set-p form)
      (set-of-p form)))

(defun tv-p (form)
  (or (conjunct-p        form)
      (equal-p           form)
      (ground-concept-p  form)
      (ground-role-p     form)
      (negation-p        form)
      (quantified-form-p form)))

(defun wml-wff-p (form)
  (or (ind-to-tv-p   form)
      (set-p         form)
      (tv-p          form)
      (speechact-p form)))
```

```
MRL-STRUCTURES.LISP
;; -*- Syntax: Common-lisp; Package: PENMAN; Mode: LISP; -*-

(in-package 'penman)

(defstruct mrl-statement
  "SPEECHACT FORMS"
  (type nil :type atom)
  (form nil :type (or mrl-formula mrl-term)))

(deftype mrl-term ()
  '(or mrl-ind mrl-set-term mrl-ind-term mrl-card-term))

(defstruct mrl-ind
  "CONSTANTS AND VARIABLES"
  (symbol nil :type atom))

(defstruct mrl-card-term
  (dummy nil :type atom)
  (set nil :type mrl-set-term))

(defstruct mrl-set-of-term
  "(SET-OF X (P X))"
  (var nil :type mrl-ind)
  (form nil :type mrl-formula))

(defstruct mrl-set-with-term
  "(SET-WITH A B)"
  (elements nil :type list))

(deftype mrl-set-term ()
  '(or mrl-set-of-term mrl-set-with-term))

(defstruct mrl-ind-term
  "IOTA AND ETA"
  (type nil :type atom)
  (var  nil :type mrl-ind)
  (form nil :type mrl-formula))

(defstruct mrl-unary-predicate
  (pred nil :type atom)
  (term nil :type mrl-term))

(defstruct mrl-binary-predicate
  (pred   nil :type atom)
  (first  nil :type mrl-term)
  (second nil :type mrl-term))

(deftype mrl-atomic-formula ()
  '(or mrl-unary-predicate mrl-binary-predicate))

(defstruct mrl-conjunction
  (components nil :type list))

(defstruct mrl-negation
  (form nil :type mrl-formula))

(defstruct mrl-quantified-formula
  "ALL, EXIST, EXIST1!, EXIST1?, EXIST>1?"
  (quantifier nil :type atom)
  (var        nil :type mrl-ind)
  (range      nil :type mrl-formula)
  (pred       nil :type mrl-formula))

(deftype mrl-formula ()
  '(or mrl-atomic-formula
      mrl-conjunction
      mrl-negation
      mrl-quantified-formula))

(defun mrl-parse (form)

  (cond
    ((atom form) (make-mrl-ind :symbol form))
    (t (case (first form)
             ((query assert command answer)
              (make-mrl-statement
                :type (first form)
                :form (mrl-parse (second form))))
             (set-of
               (make-mrl-set-of-term
                 :var (mrl-parse (second form))
                 :form (mrl-parse (third form))))
             (set-with
               (make-mrl-set-with-term
                 :elements (for elt in (rest form)
                                collect (mrl-parse elt))))
             (card
               (make-mrl-card-term
                 :dummy (second form)
                 :set (mrl-parse (third form))))
             ((iota eta)
              (make-mrl-ind-term
                :type (first form)
                :var  (mrl-parse (second form))
                :form (mrl-parse (third form))))
             ((exist all exist1! exist1? exist>1?)
              (make-mrl-quantified-formula
                :quantifier (first form)
                :var (mrl-parse (second form))
                :range (mrl-parse (third form))
                :pred (mrl-parse (fourth form))))
             (and
               (make-mrl-conjunction
                 :components (for sub-form in (rest form)
                                  collect
                                   (mrl-parse sub-form))))
             (not
               (make-mrl-negation
                 :form (mrl-parse (second form))))
```

```
          (otherwise (cond
                       ((and (eql (length form) 2)
                             (or (nikl:nkgetnamedconcept
                                   (intern (symbol-name
                                             (first form))
                                    kl2:*nikl-data-pkg*))
                                 (member (first form)
                                  '(processoccurrence
                                    future presperf
                                    pastperf))))
                        (make-mrl-unary-predicate
                          :pred (first form)
                          :term
                             (mrl-parse (second form))))
                       ((and (eql (length form) 3)
                             (or (nikl:nkgetnamedconcept
                                   (intern
                                    (symbol-name
                                    (first form))
                                    kl2:*nikl-data-pkg*))
                                 (member (first form)
                                      '(timeofoccurrence
                                         records domain
                                         range))))
                        (make-mrl-binary-predicate
                          :pred (first form)
                          :first (mrl-parse
                                       (second form))
                          :second (mrl-parse
                                       (third form))))
                       (t (warning
                           (format nil
                            "Cannot parse wml form:  ∀A"
                            form))))))))

(defun mrl-syn (mrl-struct)

  (typecase mrl-struct
    (mrl-statement
      (list (mrl-statement-type mrl-struct)
            (mrl-syn (mrl-statement-form mrl-struct))))
    (mrl-ind
      (mrl-ind-symbol mrl-struct))
    (mrl-card-term
  (list 'card
        (mrl-card-term-dummy mrl-struct)
        (mrl-syn (mrl-card-term-set mrl-struct))))
(mrl-set-of-term
  (list 'set-of
        (mrl-syn (mrl-set-of-term-var mrl-struct))
        (mrl-syn (mrl-set-of-term-form mrl-struct))))
(mrl-set-with-term
  (cons 'set-with
        (for elt in
             (mrl-set-with-term-elements mrl-struct)
             collect (mrl-syn elt))))
(mrl-ind-term
  (list (mrl-ind-term-type mrl-struct)
        (mrl-syn (mrl-ind-term-var mrl-struct))
        (mrl-syn (mrl-ind-term-form mrl-struct))))
(mrl-unary-predicate
  (list (mrl-unary-predicate-pred mrl-struct)
        (mrl-syn
            (mrl-unary-predicate-term mrl-struct))))
(mrl-binary-predicate
  (list (mrl-binary-predicate-pred mrl-struct)
        (mrl-syn
            (mrl-binary-predicate-first  mrl-struct))
        (mrl-syn (mrl-binary-predicate-second
                   mrl-struct))))
(mrl-conjunction
  (cons 'and (for conj in
                  (mrl-conjunction-components mrl-struct)
                   collect (mrl-syn conj))))
(mrl-negation
  (list 'not (mrl-syn (mrl-negation-form mrl-struct))))
(mrl-quantified-formula
  (list (mrl-quantified-formula-quantifier mrl-struct)
        (mrl-syn
            (mrl-quantified-formula-var mrl-struct))
        (mrl-syn
            (mrl-quantified-formula-range mrl-struct))
        (mrl-syn
            (mrl-quantified-formula-pred mrl-struct))))
(t (cond ((eq (first mrl-struct) (quote quote))
           mrl-struct)
          (t (warning
               (format nil
        "Cannot construct MRL statement from structure:  ~A"
                   mrl-struct)))))))


WML-PARSE.LISP
;; -*- Syntax: Common-lisp; Package: PENMAN; Mode: LISP; -*-

(in-package 'penman)

(defun wml-parse (form)

  (cond
    ((atom form)
     (cond ((and (nikl:nkgetnamedconcept
                  (intern (symbol-name form)
                   kl2:*nikl-data-pkg*))
                 (not (nikl:nkindividualconceptp
                        (nikl:nkgetnamedconcept
                          (intern (symbol-name form)
                           kl2:*nikl-data-pkg*)))))
            (make-pred :symbol form))
           (t (make-ind :symbol form))))
    (t
     (case (first form)
       ((query assert)
        (make-speechact :type      (first form)
                        :intension
                         (wml-parse (first (second form)))
                        :time      (second (second form))
                        :world     (third (second form))))
       (lambda
         (make-lambda :var   (make-ind :symbol
                                (first (second form)))
                      :range (wml-parse (third form))
                      :pred  (wml-parse (fourth form))))
       (equal
         (make-equal :first  (wml-parse (second form))
                     :second (wml-parse (third form))))
       (pred-to-set
         (make-pred-to-set
          :intension (wml-parse (second form))))
       (card
         (make-card :set (wml-parse (second form))))
       (set-of
   (make-set-of
    :extension (for elt in (rest form)
                    collect (wml-parse elt))))
 ((forall exists)
  (make-quantified-form :quantifier (first form)
                        :var
                          (make-ind
                             :symbol (second form))
                        :range (wml-parse
                                 (third form))
                        :pred (wml-parse
                                 (fourth form))))
 (iota
   (make-iota :var   (make-ind  :symbol (second form))
              :range (wml-parse (third form))
              :pred  (wml-parse (fourth form))))
 (sample
   (make-sample :intension (wml-parse (second form))))
 (power
   (make-power :pred (wml-parse (second form))))
 ((& and)
   (make-conjunct :components (for sub-form in
                                   (rest form)
                                   collect
                                   (wml-parse
                                          sub-form))))
 (not
   (make-negation :form (wml-parse (second form))))
 (intension
   (make-intension :form (wml-parse (second form))))
 ((present past future presperf pastperf)
  (make-tense :type (first form)
              :intension (wml-parse (second form))))
 (otherwise (cond
              ((and (eql (length form) 2)
                    (nikl:nkgetnamedconcept
                      (intern
                       (symbol-name (first form))
                       kl2:*nikl-data-pkg*)))
               (make-ground-concept
                :predicate
                 (make-pred :symbol (first  form))
                :ind (wml-parse (second form))))
              ((and (eql (length form) 3)
                        (nikl:nkgetnamedconcept
                              (intern (symbol-name
                                       (first form))
                              kl2:*nikl-data-pkg*)))
                   (make-ground-role
                    :predicate (make-pred :symbol
                                           (first form))
                    :domain    (wml-parse (second form))
                    :range     (wml-parse (third  form))))
                  (t (warning
                      (format nil
                       "Form did not parse as WML: ~A"
                       form))))))))


(defun wml-synthesize (wml-struct)

;;;Reconstructs the wml wff as a list.
;;;EQUAL provides a test with the original
;;;for accuracy of the structure-forming parse.

  (typecase wml-struct
    (speechact
      (list (speechact-type wml-struct)
            (list (wml-synthesize
                   (speechact-intension wml-struct))
                  (speechact-time wml-struct)
                  (speechact-world wml-struct))))
    (ind
      (ind-symbol wml-struct))
    (lambda
      (list 'lambda
       (list (wml-synthesize (lambda-var wml-struct)))
       (wml-synthesize (lambda-range wml-struct))
       (wml-synthesize (lambda-pred  wml-struct))))
    (pred
      (pred-symbol wml-struct))
    (equal
      (list 'equal
            (wml-synthesize (equal-first  wml-struct))
            (wml-synthesize (equal-second wml-struct))))
    (pred-to-set
      (list 'pred-to-set
       (wml-synthesize (pred-to-set-intension wml-struct))))
```

```
(card
  (list 'card (wml-synthesize (card-set wml-struct))))
(set-of
  (cons 'set-of (set-of-extension wml-struct)))
(quantified-form
  (list (quantified-form-quantifier wml-struct)
        (wml-synthesize
            (quantified-form-var wml-struct))
        (wml-synthesize
            (quantified-form-range wml-struct))
        (wml-synthesize
            (quantified-form-pred wml-struct))))
(iota
  (list 'iota
        (wml-synthesize (iota-var   wml-struct))
        (wml-synthesize (iota-range wml-struct))
        (wml-synthesize (iota-pred  wml-struct))))
(sample
  (list 'sample
   (wml-synthesize (sample-intension wml-struct))))
(power
  (list 'power (power-pred wml-struct)))
(conjunct
  (cons '& (for conjunct in
                (conjunct-components wml-struct)
                collect (wml-synthesize conjunct))))
(negation
  (list 'not
        (wml-synthesize (negation-form wml-struct))))
(intension
  (list 'intension
        (wml-synthesize (intension-form wml-struct))))
(tense
  (list (tense-type wml-struct)
        (wml-synthesize (tense-intension wml-struct))))
(ground-concept
  (list (wml-synthesize
            (ground-concept-predicate wml-struct))
        (wml-synthesize
            (ground-concept-ind       wml-struct))))
(ground-role
  (list (wml-synthesize
            (ground-role-predicate wml-struct))
        (wml-synthesize
                  (ground-role-domain    wml-struct))
              (wml-synthesize
                  (ground-role-range     wml-struct))))
      (t (warning
          (format nil "Cannot synthesize--bad WML structure")
          ))))



(defun wfl-type (s-exp)

  (cond ((atom s-exp)
         (cond ((or (eq s-exp t)
                    (eq s-exp nil))
                'tv)
               ((numberp s-exp) 'int)
               ((nikl:NKGetNamedConcept
                  (intern (symbol-name s-exp)
                          k12:*nikl-data-pkg*))
                (cond ((NiklSuperP s-exp '2-place-relation)
                       '(1 (1 tv)))
                      ((nikl:NKIndividualConceptP
                         (nikl:NKGetNamedConcept
                           (intern (symbol-name s-exp)
                                   k12:*nikl-data-pkg*))) '1)
                      (t '(1 tv))))
               (t '1)))
        (t (cond ((and (eq (first s-exp) 'lambda)
                       (eq (second (wfl-type (third s-exp)))
                           'tv))
                  '(,(first (wfl-type (third s-exp)))
                    ,(wfl-type (fourth s-exp))))
                 ((and (eq (first s-exp) 'equal)
                       (equal (wfl-type (second s-exp))
                              (wfl-type (third s-exp))))
                  'tv)
                 ((eq (first s-exp) 'pred-to-set)
                  '(set ,(first (wfl-type (second s-exp)))))
                 ((and (eq (first s-exp) 'card)
                       (eq (first (wfl-type (second s-exp)))
                           'set))
                  'int)
                 ((and (eq (first s-exp) 'set-of)
                       (for elt in (cddr s-exp)
              always (equal (wfl-type elt)
                            (wfl-type
                             (second s-exp)))))
 '(set ,(wfl-type (second s-exp))))
((and (or (eq (first s-exp) 'forall)
          (eq (first s-exp) 'exists))
      (equal (wfl-type (second s-exp))
             (first (wfl-type (
                        third s-exp))))
      (eq (second (wfl-type (third s-exp)))
          'tv)
      (eq (wfl-type (fourth s-exp))
          'tv))
 'tv)
((and (eq (first s-exp) 'iota)
      (equal (wfl-type (second s-exp))
             (first (wfl-type
                        (third s-exp))))
      (eq (second (wfl-type (third s-exp)))
          'tv)
      (eq (wfl-type (fourth s-exp))
          'tv))
 (wfl-type (second s-exp)))
((and (eq (first s-exp) 'sample)
      (equal (wfl-type (second s-exp))
             '(w (t (i (tv))))))
 '(i tv))
((and (eq (first s-exp) 'power)
      (eq
        (second (wfl-type (second s-exp)))
         'tv))
 '( ,(wfl-type (second s-exp)) tv))
((and (or (eq (first s-exp) '&)
          (eq (first s-exp) 'or)
          (eq (first s-exp) 'implies)
          (eq (first s-exp) 'iff))
      (eq (wfl-type (second s-exp)) 'tv)
      (eq (wfl-type (third  s-exp)) 'tv))
 'tv)
((and (or (eq (first s-exp) 'not)
          (eq (first s-exp) 'necessarily)
          (eq (first s-exp) 'possibly))
      (eq (wfl-type (second s-exp)) 'tv))
 'tv)
```

```
                  ((eq (first s-exp) 'intension)
                   `(w (t ,(wfl-type (second s-exp)))))
                  ((and (or (eq (first s-exp) 'present)
                            (eq (first s-exp) 'past)
                            (eq (first s-exp) 'future)
                            (eq (first s-exp) 'presperf)
                            (eq (first s-exp) 'pastperf))
                        (eq (first (wfl-type (second s-exp)))
                            'w)
                        (eq (first
                             (second
                              (wfl-type (second s-exp))))
                            't))
                   (second
                     (second (wfl-type (second s-exp)))))
                  ((and (equal (length s-exp) 2)
                        (equal
                         (first (wfl-type (first s-exp)))
                              (wfl-type (second s-exp))))
                   (second (wfl-type (first s-exp))))
                  ((and (equal (length s-exp) 3)
                        (equal (first (wfl-type
                                        `( ,(first s-exp)
                                          ,(second s-exp))))
                              (wfl-type (third s-exp))))
                   (second (wfl-type `( ,(first s-exp)
                                        ,(second s-exp)))))
                  (t (print (format nil "No type for ~a."
                                s-exp))
                     nil)))))

WML-MRL-RULES.LISP
;; -*- Syntax: Common-lisp; Package: PENMAN; Mode: LISP; -*-

(in-package 'penman)

(proclaim '(special elter inder occer reler timer))
(proclaim '(special *example*))

(defun mrl-trans (*example*)

;;;converts the demand-mrl-form of an example into a logical
;;;form for that example by
;;;parsing the form and then re-constructing that result.
```

```
    (setf (penexample-logicalform
                (gethash *example* penexamples))
          (mrl-syn
            (wml-transform
              (wml-parse (penexample-demand-mrl-form
                           (gethash *example*
                                 penexamples)))))))

(defun wml-transform (wml-struct)

;;;Map a wml structure into its mrl equivalent.

  (declare (special *current-vars* *lambda-vars*
                    *process-type* *tense* *vars-alist*))

  (typecase wml-struct
    (speechact
      (for obj in '(*current-vars* *lambda-vars*
                    *process-type* *tense* *vars-alist*)
           do (set obj nil))
      (make-mrl-statement :type (speechact-type wml-struct)
                          :form
                           (wml-transform
                            (speechact-intension
                               wml-struct))))
    (ind
    ;;;Use the symbol unless it has an associated lambda-var
      (if (and (eq *process-type* 'event)
               (eq (ind-symbol wml-struct) t))
          (throw 'no-event nil)
          (make-mrl-ind :symbol
                         (cond ((rest (assoc (ind-symbol
                                               wml-struct)
                                          *vars-alist*)))
                               (t (ind-symbol wml-struct))))))
    (lambda
     ;;;any LAMBDA embeds *current-vars* throughout its form
      (let ((conjunction (make-mrl-conjunction)))
        (push (cons (ind-symbol (lambda-var wml-struct))
                    (first *current-vars*)) *vars-alist*)
        (push (ind-symbol (lambda-var wml-struct))
              *lambda-vars*)
        (or
          (and
        (catch (case *process-type*
                 (event    'no-event)
                 (relation 'no-relation)
                 (class    'no-class)
                 (t        'no-process))
          (setf (mrl-conjunction-components conjunction)
                (list (wml-transform (lambda-pred
                                      wml-struct))))
        (push (transform-lambda-range wml-struct)
              (mrl-conjunction-components conjunction)))
      (and (catch (case *process-type*
                    (event    'no-event)
                    (relation 'no-relation)
                    (class    'no-class)
                    (t        'no-process))
             (setf (mrl-conjunction-components
                             conjunction)
                   (list (transform-lambda-range
                             wml-struct))))
           (setf
             (mrl-conjunction-components conjunction)
                 (append (mrl-conjunction-components
                             conjunction)
                         (list
                          (wml-transform
                           (lambda-pred wml-struct))))))
      (and
        (pop *vars-alist*)
        (pop *lambda-vars*)
        (throw (case *process-type*
                 (event    'no-event)
                 (relation 'no-relation)
                 (class    'no-class)
                 (t        'no-process))
          nil)))
    (pop *vars-alist*)
    (pop *lambda-vars*)
    conjunction))
(pred
  (pred-symbol wml-struct))
(equal
  (if (eq *process-type* 'relation)
      (make-occurrence-envelope wml-struct)
      (make-mrl-binary-predicate
          :pred   'identity
          :first  (wml-transform (equal-first wml-struct))
          :second (wml-transform (equal-second
                       wml-struct)))))
(pred-to-set
  (typecase (lambda-range (pred-to-set-intension
                               wml-struct))
    ((or lambda power)
     (prog1
      (make-mrl-set-of-term
        :var
         (make-mrl-ind
          :symbol
           (first (push (genelt) *current-vars*)))
        :form
         (wml-transform
          (pred-to-set-intension wml-struct)))
      (pop  *current-vars*)))
    (otherwise
      (make-mrl-ind-term
        :type 'iota
        :var
         (make-mrl-ind
          :symbol
           (first (push (genind) *current-vars*)))
        :form
         (wml-transform
          (pred-to-set-intension wml-struct))))))
(card
  (make-mrl-card-term
    :dummy 'fubar
    :set (wml-transform (card-set wml-struct))))
(set-of
  (make-mrl-set-with-term
    :elements (for elt in (set-of-extension wml-struct)
                    collect (wml-transform elt))))
(quantified-form
  (if (and (eq *process-type* 'event)
           (pred-p (quantified-form-range wml-struct))
           (niklsuperp
            (pred-symbol
             (quantified-form-range wml-struct))
            'event))
      (make-occurrence-envelope wml-struct)
```

```
(let ((qform (make-mrl-quantified-formula
               :quantifier
               (case
                 (quantified-form-quantifier
                  wml-struct)
                 (forall 'all)
                 (exists (typecase
                           (quantified-form-range
                            wml-struct)
                           (sample 'exist>1?)
                           (t 'exist))))
               :var (make-mrl-ind
                      :symbol
                       (ind-symbol
                         (quantified-form-var
                          wml-struct))))))
  (if (and (ind-p (quantified-form-pred
                    wml-struct))
           (eq (ind-symbol
                 (quantified-form-pred wml-struct))
               t)
           (eq *process-type* 'relation))
      (and (setf
             (mrl-quantified-formula-pred qform)
               (make-occurrence-envelope
                 (quantified-form-pred wml-struct)
                   (quantified-form-var
                     wml-struct)))
           (setf (mrl-quantified-formula-range
                    qform)
                 (transform-quantified-formula-range
                   wml-struct)))
      (or(and
           (catch (case *process-type*
                     (event    'no-event)
                     (relation 'no-relation)
                     (class    'no-class)
                     (t        'no-process))
             (setf (mrl-quantified-formula-pred
                      qform)
                    (wml-transform
                     (quantified-form-pred
                               wml-struct))))
           (setf (mrl-quantified-formula-range
                            qform)
                     (transform-quantified-formula-range
                         wml-struct)))
               (and
                 (catch (case *process-type*
                          (event    'no-event)
                          (relation 'no-relation)
                          (class    'no-class)
                          (t        'no-process))
                   (setf (mrl-quantified-formula-range
                            qform)
                     (transform-quantified-formula-range
                           wml-struct)))
                 (setf
                   (mrl-quantified-formula-pred qform)
                       (wml-transform
                        (quantified-form-pred
                         wml-struct))))
               (throw (case *process-type*
                        (event    'no-event)
                        (relation 'no-relation)
                        (class    'no-class)
                        (t        'no-process))
                 nil)))
        qform)))
(iota
  (typecase (iota-range wml-struct)
    (power (make-mrl-set-of-term
             :var (make-mrl-ind
                    :symbol (ind-symbol
                              (iota-var wml-struct)))
             :form (make-mrl-conjunction
                     :components
                     (list (make-mrl-unary-predicate
                             :pred
                              (pred-symbol
                               (power-pred
                                (iota-range wml-struct))
                             :term
                              (wml-transform
                               (iota-var wml-struct)))
                           (wml-transform
                            (iota-pred wml-struct))))))
    (t (make-mrl-ind-term
          :type 'iota
          :var  (make-mrl-ind
                     :symbol
                      (ind-symbol
                       (iota-var wml-struct)))
          :form (make-mrl-conjunction
                  :components
                  (list (make-mrl-unary-predicate
                              :pred
                               (pred-symbol
                                (iota-range wml-struct))
                              :term
                               (wml-transform
                                (iota-var wml-struct)))
                        (wml-transform
                         (iota-pred wml-struct)))))))
(sample
  (wml-transform (sample-intension wml-struct)))
(power
  (wml-transform (power-pred wml-struct)))
(conjunct
  (make-mrl-conjunction
    :components (for conj in (conjunct-components
                                wml-struct)
                  collect (wml-transform conj))))
(negation
  (make-mrl-negation
    :form (wml-transform (negation-form wml-struct))))
(intension
  (wml-transform (intension-form wml-struct)))
(tense
  (case *process-type*
    (event    (throw 'no-event    nil))
    (relation (throw 'no-relation nil))
    (class    (throw 'no-class    nil))
    (otherwise
      (setf *tense* (tense-type wml-struct))
      (or (and (setf *process-type* 'event)
               (catch 'no-event
                      (wml-transform
                       (tense-intension wml-struct))))
          (and (setf *process-type* 'relation)
               (catch 'no-relation
                      (wml-transform
                     (tense-intension wml-struct)))
          (and (setf *process-type* 'class)
               (catch 'no-class
                      (wml-transform
                       (tense-intension wml-struct)))
          (warning
           (format nil
            "No process identified for time operator ~a"
            *tense*))))))
(ground-concept
  (cond ((eq *process-type* 'event)
         (throw 'no-event nil))
        ((eq *process-type* 'relation)
         (throw 'no-relation nil))
        ((eq *process-type* 'class)
         (make-occurrence-envelope wml-struct))
        (t (make-mrl-unary-predicate
             :pred
              (pred-symbol
               (ground-concept-predicate wml-struct))
             :term
              (wml-transform
               (ground-concept-ind wml-struct)))))))
(ground-role
  (cond ((eq *process-type* 'event)
         (throw 'no-event nil))
        ((eq *process-type* 'relation)
         (make-occurrence-envelope wml-struct))
        ((niklsuperp (pred-symbol
                      (ground-role-predicate wml-struct)
                     'participantrelation)
         (make-mrl-binary-predicate
           :pred
            (pred-symbol
             (ground-role-predicate wml-struct))
           :first
             (wml-transform
              (ground-role-domain wml-struct))
           :second
            (wml-transform
             (ground-role-range wml-struct))))
        (t (let ((rel (make-mrl-ind :symbol (genrel)))
             (make-mrl-quantified-formula
             :quantifier 'exist1!
```

```
                 :var rel
                 :range
                  (make-mrl-unary-predicate
                   :pred
                    (pred-symbol
                     (ground-role-predicate wml-struct))
                   :term rel)
                 :pred
                  (make-mrl-conjunction
                   :components
                    (list
                     (make-mrl-binary-predicate
                        :pred 'domain
                        :first rel
                        :second
                         (wml-transform
                          (ground-role-domain wml-struct)))
                           (make-mrl-binary-predicate
                                :pred 'range
                                :first rel
                                :second
                                 (wml-transform
                                  (ground-role-range
                                   wml-struct)))))
                 )))))))


(defun transform-lambda-range (wml-struct)

  (typecase (lambda-range wml-struct)
    (pred (make-mrl-unary-predicate
            :pred
             (pred-symbol
              (lambda-range  wml-struct))
            :term
             (make-mrl-ind
              :symbol
               (first *current-vars*))))
    (lambda
      (push (cons (ind-symbol
                    (lambda-var (lambda-range wml-struct)))
                   (first *current-vars*))
       *vars-alist*)
      (push (ind-symbol (lambda-var (lambda-range
                                            wml-struct)))
                *lambda-vars*)
          (prog1
            (make-mrl-conjunction
              :components
              (list (make-mrl-unary-predicate
                      :pred
                       (pred-symbol
                        (lambda-range
                         (lambda-range wml-struct)))
                      :term  (make-mrl-ind
                              :symbol (first *current-vars*)))
                    (wml-transform
                     (lambda-pred (lambda-range wml-struct)))))
            (pop *lambda-vars*)
            (pop *vars-alist*)))
        (power
          (typecase (power-pred (lambda-range wml-struct))
            (pred (make-mrl-unary-predicate
                    :pred
                     (pred-symbol
                      (power-pred (lambda-range wml-struct)))
                    :term
                     (make-mrl-ind
                      :symbol (first *current-vars*))))
            (t (wml-transform (lambda-range wml-struct))))
          )))

(defun transform-quantified-formula-range (wml-struct)

  (typecase (quantified-form-range wml-struct)
    ((or lambda sample)
     (push (ind-symbol (quantified-form-var wml-struct))
           *current-vars*)
     (prog1 (wml-transform
              (quantified-form-range wml-struct))
            (pop  *current-vars*)))
    (power (make-mrl-unary-predicate
             :pred
              (power-pred
               (quantified-form-range wml-struct))
             :term
              (wml-transform
                (quantified-form-var wml-struct)))))
    (pred  (make-mrl-unary-predicate
              :pred (pred-symbol
                       (quantified-form-range wml-struct))
              :term (wml-transform
                       (quantified-form-var wml-struct))))))
                                                                --|
(defun make-occurrence-envelope
       (struct &optional var-struct)

  (let ((occ   (make-mrl-ind :symbol (genocc))
        (time  (make-mrl-ind :symbol (gentim)))
        (tense *tense*))

    (setf *tense* nil)
    (setf *process-type* nil)
    (typecase struct
      (quantified-form
        ;;;This case corresponds to finding an event.
        (let ((event (wml-transform (quantified-form-var
                                                   struct))))
          (make-mrl-quantified-formula
            :quantifier 'existi!
            :var occ
            :range (make-mrl-unary-predicate
                        :pred 'occurrence
                        :term occ)
            :pred
             (make-mrl-quantified-formula
                :quantifier 'exist
                :var event
                :range (make-mrl-unary-predicate
                            :pred (wml-transform
                                     (quantified-form-range
                                      struct))
                            :term event)
                :pred
                 (make-mrl-conjunction
                    :components
                     (cons
                      (make-mrl-binary-predicate
                         :pred 'timeofoccurrence
                         :first occ
                         :second
                          (make-mrl-ind-term
                       :type 'eta
                       :var time
                       :form (make-mrl-unary-predicate
                                :pred tense
                                :term time)))
                (cons
                  (make-mrl-binary-predicate
                     :pred 'records
                     :first occ
                     :second event)
                  (typecase
                   (quantified-form-pred struct)
                   (conjunct
                    (for comp in
                       (conjunct-components
                        (quantified-form-pred struct))
                       collect (wml-transform comp)))
                   (t (list (wml-transform
                              (quantified-form-pred
                                  struct)))
                                      )))))))))
((or equal ground-role ind)
  ;;;This case corresponds to finding a relation.
  (let ((rel (make-mrl-ind :symbol (genrel))))
    (make-mrl-quantified-formula
      :quantifier 'exist1!
      :var occ
      :range (make-mrl-unary-predicate
               :pred 'occurrence
               :term occ)
      :pred (make-mrl-quantified-formula
              :quantifier 'exist1!
              :var rel
              :range
              (make-mrl-unary-predicate
                :pred (typecase struct
                        (ground-role
                          (pred-symbol
                           (ground-role-predicate
                            struct)))
                        (equal 'identity)
                        (ind 'existence))
                :term rel)
              :pred
```

```
(make-mrl-conjunction
  :components
  (cons
    (make-mrl-binary-predicate
      :pred 'timeofoccurrence
      :first occ
      :second
       (make-mrl-ind-term
        :type 'eta
        :var  time
        :form
         (make-mrl-unary-predicate
          :pred tense
          :term time)))
    (cons
      (make-mrl-binary-predicate
        :pred 'records
        :first occ
        :second rel)
      (cons
        (make-mrl-binary-predicate
          :pred 'domain
          :first rel
          :second
            (wml-transform
             (typecase struct
              (ground-role
               (ground-role-domain
                           struct))
              (equal
                (equal-first struct))
              (ind  var-struct))))
        (typecase struct
          (ind nil)
          (t (list
              (make-mrl-binary-predicate
                 :pred 'range
                 :first rel
                 :second
                 (wml-transform
                   (typecase struct
                     (ground-role
                      (ground-role-range
                       struct))
```

```
                                   (equal
                                    (equal-second
                                     struct))))))))))
              )))))
(ground-concept
 ;;;This case corresponds to finding a class.
  (let ((rel (make-mrl-ind :symbol (genrel)))
        (elt (make-mrl-ind :symbol (genelt))))
    (make-mrl-quantified-formula
      :quantifier 'exist1!
      :var occ
      :range (make-mrl-unary-predicate
               :pred 'occurrence
               :term occ)
      :pred (make-mrl-quantified-formula
              :quantifier 'exist1!
              :var  rel
              :range (make-mrl-unary-predicate
                       :pred 'classascription
                       :term rel)
              :pred (make-mrl-conjunction
                      :components
                      (list
                       (make-mrl-binary-predicate
                          :pred 'timeofoccurrence
                          :first occ
                          :second
                           (make-mrl-ind-term
                              :type 'eta
                              :var  time
                              :form
                             (make-mrl-unary-predicate
                                :pred tense
                                :term time)))
                            (make-mrl-binary-predicate
                              :pred 'records
                              :first occ
                              :second rel)
                            (make-mrl-binary-predicate
                              :pred 'domain
                              :first rel
                              :second
                               (wml-transform
                                 (ground-concept-ind
                                        struct)))
                                  (make-mrl-binary-predicate
                                    :pred 'range
                                    :first rel
                                    :second
                                     (make-mrl-set-of-term
                                         :var elt
                                         :form
                                (make-mrl-unary-predicate
                                             :pred
                                              (wml-transform
                                (ground-concept-predicate
                                                 struct))
                                             :term elt)
                                        ))))))))
      )))

(defun elter (e) (function (lambda () (setf e (+ e 1)))))
(defun inder (i) (function (lambda () (setf i (+ i 1)))))
(defun occer (o) (function (lambda () (setf o (+ o 1)))))
(defun reler (r) (function (lambda () (setf r (+ r 1)))))
(defun timer (s) (function (lambda () (setf s (+ s 1)))))
(setf elter (elter 0))
(setf inder (inder 0))
(setf occer (occer 0))
(setf reler (reler 0))
(setf timer (timer 0))

(defun genrel ()
;;;return a nice gnenerated symbol for relations
    (intern (format nil "R∀A" (funcall reler))))

(defun genelt ()
;;;return a nice gnenerated symbol for relations
    (intern (format nil "E∀A" (funcall elter))))

(defun genind ()
;;;return a nice gnenerated symbol for individuals
    (intern (format nil "I∀A" (funcall inder))))

(defun genocc ()
;;;return a nice gnenerated symbol for occurrences
    (intern (format nil "O∀A" (funcall occer))))
```

```
(defun gentim ()
;;;return a nice gnenerated symbol for times
    (intern (format nil "T∇A" (funcall timer))))
```

References

[Barwise & Perry 83] Jon Barwise and John Perry, *Situations and Attitudes,* The MIT Press, 1983.

[Cooper 87] Robin Cooper, "Meaning representation in Montague grammar and situation semantics," *Computational Intelligence* 3, 1987, 35-44.

[Habel 82] Christopher Habel, "Referential nets with attributes," in Horecky (ed.), *Proceedings of COLING-82,* North-Holland, Amsterdam, 1982.

[Hinrichs et al. 87] Erhard Hinrichs, Damaris Ayuso, and Remko Scha, "The syntax and semantics of a meaning representation language for janus," in *Research and Development in Natural Language Understanding as Part of the Strategic Computing Program, BBN Annual Technical Report 6522 December 1985--December 1986*, BBN, Cambridge, MA, 1987.

[Hobbs 85] Jerry R. Hobbs, "Ontological promiscuity," in *Proceedings, 23rd Annual Meeting of the Association for Computational Linguistics*, ACL, Chicago, IL, July 1985.

[Kaczmarek et al. 86] Tom Kaczmarek, Ray Bates, and Gabriel Robins, "Recent developments in NIKL," in *AAAI-86, Proceedings of the National Conference on Artificial Intelligence*, AAAI, Philadelphia, PA, August 1986.

[Lewis 72] David Lewis, "General Semantics," in Davidson and Harman (ed.), *Semantics of Natural Languages*, D. Reidel Publishing Company, 1972.

[Mann & Matthiessen 83] William C. Mann and Christian M. I. M. Matthiessen, *Nigel: A Systemic Grammar for Text Generation*, USC/Information Sciences Institute, Technical Report ISI/RR-83-105, February 1983.

[Montague 74] Richard Montague, *Formal Philosophy,* Yale University Press, New Haven, CT, 1974.

[Sondheimer & Nebel 86] Norman Sondheimer and Bernhard Nebel, "A logical-form and knowledge-base design for natural language generation," in *AAAI-86, Proceedings of the National Conference on Artificial Intelligence*, AAAI, Philadelphia, PA, August 1986.

[Vilain 85] Marc Vilain, "The restricted language architecture of a hybrid representation system," in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 547-551, Los Angeles, CA, August 1985.

[Walker et al. 85] Ed Walker, Ralph Weischedel, and Norman Sondheimer, "Natural language interface technology," in *Strategic Systems Symposium*, DARPA, Monterey, CA, October 1985.